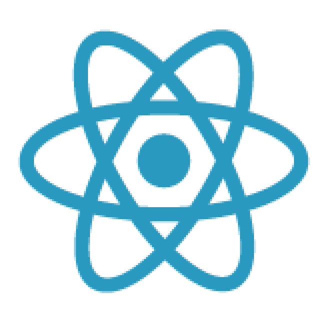


React 16
React Router
Redux
MobX

React进阶之路

• 徐 超 编著 •

React 全栈开发

清華大學出版社



React 进阶之路

·徐 超 编著·

清華大學出版社
北 京

内 容 简 介

本书详细介绍了 React 技术栈涉及的主要技术。本书分为基础篇、进阶篇和实战篇三部分。基础篇主要介绍 React 的基本用法，包括 React 16 的新特性；进阶篇深入讲解组件 state、虚拟 DOM、高阶组件等 React 中的重要概念，同时对初学者容易困惑的知识点做了介绍；实战篇介绍 React Router、Redux 和 MobX 3 个 React 技术栈的重要成员，并通过实战项目讲解这些技术如何和 React 结合使用。

本书示例丰富、注重实战，适用于从零开始学习 React 的初学者，或者已经有一些 React 使用经验，但希望更加全面、深入理解 React 技术栈的开发人员。阅读本书，需要先掌握基础的前端开发知识。

图书在版编目（CIP）数据

React 进阶之路/徐超编著. —北京：清华大学出版社，2018
ISBN 978-7-302-49801-8

Ⅰ. ①R…　Ⅱ. ①徐…　Ⅲ. ①移动终端－应用程序－程序设计　Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2018）第 037116 号

责任编辑： 王金柱
封面设计： 王　翔
责任校对： 闫秀华
责任印制： 李红英

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn，http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者： 北京密云胶印厂
经　　销： 全国新华书店
开　　本： 190mm×260mm　　**印　　张：** 15.75　　**字　　数：** 403 千字
版　　次： 2018 年 4 月第 1 版　　**印　　次：** 2018 年 4 月第 1 次印刷
印　　数： 1~3500
定　　价： 69.00 元

产品编号：077579-01

推 荐 序

小时候，老师问大家长大的理想是什么。我记得曾自豪地说——工程师。后来，真的走进了计算机领域，成为一名软件工程师。在学校里学的都是基础课，记忆犹新的有计算机原理、操作系统、编译原理、数据结构和算法等，感觉是终身受益，就像练武功，都要练好弓、马、仆、虚、歇5种基本步型一样。

那时并没有前端的说法。人机界面开始主要以程序员使用为主，通过黑洞洞的 Terminal 来编程，程序员还乐此不疲。后来出现了第一波突破——各种图形界面，PC 变得亲民。而以 iPhone 带领的移动终端的第二波革新让用户能够通过触摸、视觉和声音真正自然地与设备交互。将来必然拥有超越触摸、视觉和声音识别的技术，属于传感器和物联网的时代。这种技术使用传感器和人工智能识别身体运动、温度变化和其他环境要素，并据此做出回应，使得设备看起来可以读懂内心的想法一样。在不久的将来，一个传感器阵列能够提供高度的情境感知，并且协同工作，收集和处理关于周边环境的信息，通过人工智能预测需求并做出完全个性化的安排。前端工程师的使命也随着人机交互的显著进步而不断拓展。

时光回到刚工作时的 2000 年，正值互联网的发展初期，作为一名软件工程师，解决问题就是关键，对于前后端编程都需要熟悉。当时，前端编程的核心技能有 HTML、CSS、JavaScript， 对于习惯逻辑思维的工程师，学起来并不算难。随着互联网的发展，特别是 2010 年后，移动设备成为主流，前端工程师角色被行业认可，并且越来越重要，涵盖多终端的视觉和交互的实现，面对的是软件工程的一个持久的挑战——人机交互。首先，人机交互是软件产品里变化最频繁的部分，同时是非常关键的一环。其次，兼容各种浏览器、Web 的标准，以及适配多种终端，都是很大的挑战。另外，前端领域的技术发展也越来越快，各种新的思想、设计模式、工具和平台不断出现，怎样快速学习、在不同场景下做出恰当的选择是成为一位优秀前端工程师必备的素质。许多人机交互问题有非常巧妙的思路和精彩的解决办法。不得不说，前端工程师在工程师群体里属于非常有创造力、想象力的一群人。

前端领域各种新技术、新思想不断涌现，AngularJS、React、Vue.js、Node.js、ES 6、ES 7、CoffeeScript、TypeScript，令人眼花缭乱。对于许多开发者，估计还没学明白一样技术，就发现其已被另一些新的技术取代而“过时”了。但是，如果退一步来看，前端的基本功仍然是 HTML、CSS、JavaScript，还有算法、数据结构、编译原理。这一点，有点像《笑傲江湖》里，令狐冲一旦领悟了独孤九剑，永远能够无招胜有招。

除了具备扎实的基本功之外，一个优秀的前端工程师必须要有自己擅长的领域，并且钻研得足够深入，只有花时间学习成体系的知识才能从中总结出规律并形成方法论，从而最大化学习的价

值。同时要有广泛的视野，不能局限于前端本身，因为有很多东西只有站在前端之外才能看得更清晰、更透彻。例如，React 集成了许多后端的优秀理念，包括采用声明范式轻松描述应用、通过抽象 DOM 来达到高效的编程。围绕 React 还出现了许多工具和框架，形成了 React 生态。React 逐渐从最早的 UI 引擎变成了前后通吃的 Web App 解决方案，衍生出来的 React Native 又实现了用 Web App 的方式去写 Native App。这样，同一组人写一次 UI 就能运行在浏览器、移动终端和服务器上。

作为智能物联网先锋的远景智能，一直崇尚工程师文化和工匠精神，非常强调基本功、专业深度以及跨界创新。前端团队徐超写的《React 进阶之路》，内容由浅入深，再结合实战，很像我读大一时的 Java 101 课程的教材，对于需要学习 React 的读者是一部非常好的参考书。读这本书，最好的方法是领悟其精髓，掌握软件设计之路，灵活使用以解决问题。工程师不能因为太细的学科限制了自己的思维，也不能像大公司一个工作一个螺丝钉，在很窄的领域里重复劳动。工程师天生是发现问题、解决问题、优化问题的。达 • 芬奇、特斯拉之所以是完美的工程师，因为他们会掌握各种学科，融合并创新，在解决问题的同时开创先河。未来的信息化世界就是要不断地聪明学习，融合各种学科，通过实践解决问题，奇思妙想地创造技术的进步。

计算机的不同语言、不同技术和算法就好比一堆便宜或者昂贵的工具（如锥子和刨子），其实这些都不重要，因为大家都忽略了，做出漂亮器具的是那个工匠，而不是工具。脑子里的经验积累、天赋、执着与认真的态度、不停尝试、追求完美的态度，加起来才能创造好的作品与产品。计算机语言就像赛车场上的跑车，换了车队和跑车，舒马赫还是 F1 车神，观众还是会为其欢呼雀跃，正因为车神掌握了与跑车和赛道的沟通之道！

远景智能技术副总裁、前阿里巴巴集团淘宝 CTO

余海峰

前　　言

当今，前端应用需要解决的业务场景正变得越来越复杂，这也直接推动了前端技术的迅速发展，各种框架和类库日新月异、层出不穷。面对众多的框架和类库，前端开发者可能感到眼花缭乱，但换一个角度来看，这未尝不是一种百家争鸣的现象。不同框架和类库的设计思想和设计理念各有千秋，解决的问题也有所不同，这些多元化和差异化不断推动前端技术的发展，同时也是前端技术领域的一份思想瑰宝。

React 作为当今众多新技术的一个代表，由 Facebook 开源，致力于解决复杂视图层的开发问题，它提出一种全新的 UI 组件的开发理念，降低了视图层的开发复杂度，提高了视图层的开发效率，让页面开发变得简单、高效、可控。此外，React 不仅是单一的类库，更是一个技术栈生态，可以和生态中的 Redux、MobX 等其他技术结合使用，构建可扩展、易维护、高性能的大型 Web 应用。

本书内容

本书涵盖 React 技术栈中的主要技术，内容由浅到深。本书内容分为基础篇、进阶篇和实战篇，每一篇内容又分成若干章节来介绍。

基础篇，介绍了 React 的基本概念，包括 React 的开发环境和开发工具、React 的基本用法和 React 16 的新特性。每个知识点都有配套的项目示例。

进阶篇，深入介绍了 React 的几个重要概念，如组件 state、虚拟 DOM、高阶组件等，此外，还针对初学者使用 React 时容易产生困惑的知识点做了专门讲解，如组件与服务器通信、组件之间通信、组件的 ref 属性等。

实战篇，介绍了 React 技术栈中最重要的三个技术：React Router、Redux 和 MobX，每一个技术都配有详细的项目实战示例。

本书章节的难度逐步递增，各章节的知识存在依赖关系，所以读者需按照章节顺序阅读本书，不要随意跳跃章节，尤其是在阅读实战篇时，务必保证已经掌握了基础篇和进阶篇的内容，否则，阅读实战篇可能会有些吃力。

本书特点

本书的特点是内容全、知识新、实战性强。

内容全：本书不仅详细介绍了 React 的使用，还详细介绍了 React 技术栈中最常用的其他相关技术：React Router、Redux 和 MobX。

知识新：本书介绍的知识点都是基于各个框架、类库当前的最新版本，尤其是涵盖 React

16 的新特性和 React Router 4 的介绍。对于新版本已经不再支持或建议废弃的特性，本书不会再介绍，确保读者所学知识的时效性。

实战性强：本书配有大量示例代码，保证读者学以致用。实战篇使用的简易 BBS 项目示例接近真实项目场景，但又有所简化，让读者既可以真正理解和领会相关技术在真实项目中的使用方式，又不会因为示例项目过于复杂而影响学习。

本书目标读者

本书面向希望从零开始学习 React 的初学者，或者已经有一些 React 使用经验，希望更加全面、深入理解 React 技术栈的开发人员。

示例代码

本书的示例代码下载地址为 https://github.com/xuchaobei/react-book。如果读者发现代码或者书中的错误，可以直接在该代码仓库提交 issue。

本书中默认的开发环境是 Node.js v8.4.0，书中介绍到的几个主要库的版本分别为 React 16.1.1、React Router 4.2.2、Redux 3.7.2 及 MobX 3.3.1。

致谢

本书的完成离不开在各个方面给过我支持和帮助的人，请允许我在这里向他们表示感谢。

首先，感谢公司的领导余海峰（Colin）和贺鸣（Sky）对我写书的支持。Colin 在百忙之中还抽出时间为本书作序。

其次，感谢我的同事王博、陈小梦、吴福城、詹敏和朱雅琴，他们给本书提出了很多宝贵的意见。

还要感谢我的老婆，2017 年，她的新书《时间的格局：让每一分钟为未来增值》出版，这也让我产生了写书的念头，同时她的写书经验也给了我很多帮助。

最后，感谢清华大学出版社的王金柱老师，正是缘于他的主动联系，才让我写书的念头变成了行动。他认真、负责的工作态度也保证了本书的顺利问世。

联系作者

欢迎各位读者通过我的微信订阅号：老干部的大前端（ID：Broad_FE）和我进行沟通交流，订阅号还提供了更多的大前端学习资源。读者可以扫描右方二维码关注订阅号。



徐 超

2018 年 1 月 1 日于上海

目　　录

第 1 篇　基础篇——React，一种革命性的 UI 开发理念

第 2 篇 进阶篇——用好 React，你必须要知道的那些事

第 3 篇　实战篇——在大型 Web 应用中使用 React

第 1 篇　基础篇

React，一种革命性的UI开发理念

第 1 章

初识 React

当今，Web 应用的业务场景正在变得越来越复杂，几乎所有应用都在尝试或者已经在 Web 上使用，同时，用户对 Web 应用的体验要求也越来越高，这一切都给前端开发人员开发前端界面带来了巨大的挑战。而 Facebook 开源的 React 技术创造性地提出了一种全新的 UI 开发理念，让 UI 开发变得简单、高效、可控。React 自开源以来，迅速风靡整个前端世界，推动前端开发有了革命性的进步。

1.1 React 简介

前端 UI 的本质问题是如何将来源于服务器端的动态数据和用户的交互行为高效地反映到复杂的用户界面上。React 另辟蹊径，通过引入虚拟 DOM、状态、单向数据流等设计理念，形成以组件为核心，用组件搭建 UI 的开发模式，理顺了 UI 的开发过程，完美地将数据、组件状态和 UI 映射到一起，极大地提高了开发大型 Web 应用的效率。

React 的特点可以归结为以下 4 点：

（1）声明式的视图层。使用 React 再也不需要担心数据、状态和视图层交错纵横在一起了。React 的视图层是声明式的，基于视图状态声明视图形式。但 React 的视图层又不同于一般的 HTML 模板，它采用的是 JavaScript（JSX）语法来声明视图层，因此可以在视图层中随意使用各种状态数据。

（2）简单的更新流程。React 声明式的视图定义方式有助于简化视图层的更新流程。你只需要定义 UI 状态，React 便会负责把它渲染成最终的 UI。当状态数据发生变化时，React 也会根据最新的状态渲染出最新的 UI。从状态到 UI 这一单向数据流让 React 组件的更新流程清晰简洁。

（3）灵活的渲染实现。React 并不是把视图直接渲染成最终的终端界面，而是先把它们渲染成虚拟 DOM。虚拟 DOM 只是普通的 JavaScript 对象，你可以结合其他依赖库把这个对象渲染成不同终端上的 UI。例如，使用 react-dom 在浏览器上渲染，使用 Node 在服务器端渲染，使用 React Native 在手机上渲染。本书主要以 React 在浏览器上的渲染为例介绍 React 的使用，但你依然可以很容易地把本书的知识应用到 React 在其他终端的渲染上。

（4）高效的 DOM 操作。我们已经知道虚拟 DOM 是普通的 JavaScript 对象，正是有了虚拟 DOM 这一隔离层，我们再也不需要直接操作又笨又慢的真实 DOM 了。想象一下，操作一个 JavaScript 对象比直接操作一个真实 DOM 在效率上有多么巨大的提升。而且，基于 React 优异的差异比较算法，React 可以尽量减少虚拟 DOM 到真实 DOM 的渲染次数，以及每次渲染需要改变的真实 DOM 节点数。

虽然 React 有这么多强大的特性，但它并不是一个 MVC 框架。从 MVC 的分层来看，React 相对于 V 这一层，它关注的是如何根据状态创建可复用的 UI 组件，如何根据组件创建可组合的 UI。当应用很复杂时，React 依然需要结合其他库（如 Redux、MobX 等）使用才能发挥最大作用。

1.2 ES 6 语法简介

ES 6 是 JavaScript 语言的新一代标准，加入了很多新的功能和语法。React 的项目一般都是用 ES 6 语法来写的，这也是 Facebook 官方推荐的方式。为保证本书知识体系的完整性，本节我们会对开发 React 应用经常用到的 ES 6 语法做简要介绍。

1. let、const

let 和 const 是 ES 6 中新增的两个关键字，用来声明变量，let 和 const 都是块级作用域。let 声明的变量只在 let 命令所在的代码块内有效。const 声明一个只读的常量，一旦声明，常量的值就不能改变。例如：

```
// let 示例
{
  var a = 1;
  let b = 2;
}
a          // 1
b          // ReferenceError: b is not defined.

//const 示例
const c = 3;
c = 4;      //TypeError: Assignment to constant variable.
```

2. 箭头函数

ES 6 允许使用“箭头”（=>）定义函数。这种方式创建的函数不需要 function 关键字，并且

还可以省略 return 关键字。同时，箭头函数内的 this 指向函数定义时所在的上下文对象，而不是函数执行时的上下文对象。例如：

```
var f = a => a + 1;
//等价于
var f = function(a){
  return a + 1;
}

function foo(){
  this.bar = 1;
  this.f = (a) => a + this.bar;
}
//等价于
function foo(){
  this.bar = 1;
  this.f = (function(a){
    return a + this.bar
  }).bind(this);
}
```

如果箭头函数的参数多于 1 个或者不需要参数，就需要使用一个圆括号代表参数部分。例如：

```
var f = () => 1;
var f = (a, b) => a + b;
```

如果函数体内包含的语句多于一条，就需要使用大括号将函数体括起来，使用 return 语句返回。例如：

```
var f = (x, y) => {
  x++;
  y--;
  return x + y;
}
```

3. 模板字符串

模板字符串是增强版的字符串，用反引号（`）标识字符串。除了可以当作普通字符串使用外，它还可以用来定义多行字符串，以及在字符串中嵌入变量，功能很强大。例如：

```
//普通字符串
`React is wonderful !`

//多行字符串
'JS is wonderful !
React is wonderful!'
```

```
//字符串中嵌入变量
var name = "React";
'Hello, ${name} !';
```

4. 解构赋值

ES 6 允许按照一定模式从数组和对象中提取值，对变量进行赋值，这被称为解构。例如：

```
//数组解构
let [a,b,c] = [1, 2, 3];
a //1
b //2
c //3

//对象解构
let name = 'Lily';
let age = 4;
let person = {name, age};
person // Object {name: "Lily", age: 4}

//对象解构的另一种形式
let person = {name: 'Lily', age: 4};
let {name, age} = person;
name // "Lily"
age  //4
```

函数的参数也可以使用解构赋值。例如：

```
//数组参数解构
function sum ([x, y]) {
  return x + y;
}
sum([1, 2]); // 3

//对象参数解构
function sum ({x, y}) {
  return x + y;
}
sum({x:1, y:2}); // 3
```

解构同样适用于嵌套结构的数组或对象。例如：

```
//嵌套结构的数组解构
let [a, [b], c] = [1, [2], 3];
a;  //1
b;  //2
c;  //3
```

```
//嵌套结构的对象解构
let {person: {name, age}, foo} = {person: {name: 'Lily', age: 4}, foo: 'foo'};
name  //"Lily"
age   //4
foo   //"foo"
```

5. rest 参数

ES 6 引入 rest 参数（形式为...变量名）用于获取函数的多余参数，以代替 arguments 对象的使用。rest 参数是一个数组，数组中的元素是多余的参数。注意，rest 参数之后不能再有其他参数。例如：

```
function languages(lang, ...types){
  console.log(types);
}
languages('JavaScript', 'Java', 'Python');   //["Java", "Python"]
```

6. 扩展运算符

扩展运算符是三个点（...），它将一个数组转为用逗号分隔的参数序列，类似于 rest 参数的逆运算。例如：

```
function sum(a, b, c){
  return a + b + c;
}
let numbers = [1, 2, 3];
sum(...numbers);    //6
```

扩展运算符还常用于合并数组以及与解构赋值结合使用。例如：

```
//合并数组
let arr1 = ['a'];
let arr2 = ['b', 'c'];
let arr3 = ['d', 'e'];
[...arr1, ...arr2, ...arr3];    //['a', 'b', 'c', 'd', 'e'];

//与解构赋值结合
let [a, ...rest] = ['a', 'b', 'c'];
rest   //['b', 'c']
```

扩展运算符还可以用于取出参数对象的所有可遍历属性，复制到当前对象之中。例如：

```
let bar = {a: 1, b: 2};
let foo = {...bar};
foo            //Object {a: 1, b: 2};
foo === bar  //false
```

7. class

ES 6 引入了 class（类）这个概念，新的 class 写法让对象原型的写法更加清晰，也更像传统的面向对象编程语言的写法。例如：

```
//定义一个类
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  getName(){
    return this.name;
  }

  getAge(){
    return this.age;
  }
}

//根据类创建对象
let person = new('Lily', 4);
```

class 之间可以通过 extends 关键字实现继承，例如：

```
class Man extends Person {
  constructor(name, age) {
    super(name, age);
  }

  getGender() {
    return 'male';
  }
}

let man = new Man('Jack', 20);
```

8. import、export

ES 6 实现了自己的模块化标准，ES 6 模块功能主要由两个关键字构成：export 和 import。export 用于规定模块对外暴露的接口，import 用于引入其他模块提供的接口。例如：

```
//a.js，导出默认接口和普通接口
const foo = () => 'foo';
const bar = () => 'bar';
```

```
export default foo;   //导出默认接口
export {bar};         //导出普通接口

//b.js(与 a.js 在同一目录下)，导入 a.js 中的接口
//注意默认接口和普通接口导入写法的区别
import foo, {bar} from './a';
foo();    //"foo"
bar();    //"bar"
```

本节介绍的 ES 6 语法是后面我们介绍 React 时经常用到的语法，且只介绍了最基本的用法，如果读者想了解 ES 6 完整的语法知识，请自行查阅相关文档学习。

1.3　开发环境及工具介绍

“工欲善其事，必先利其器”。开发 React 应用需要一个由众多开发工具组建成的开发环境，这个复杂的开发环境也大大提高了应用的开发和调试效率。本节将简要介绍本书使用到的相关开发工具。

1.3.1　基础环境

1.　Node.js

Node.js 是一个 JavaScript 运行时，它让 JavaScript 在服务器端运行成为现实。React 应用的执行并不依赖于 Node.js 环境，但 React 应用开发编译过程中用到的很多依赖（例如 NPM、Webpack 等）都是需要 Node.js 环境的。所以，在开发 React 应用前，需要先安装 Node.js。Node.js 的官方下载地址为 https://nodejs.org。本书使用的 Node.js 的版本号为 v8.4.0，建议读者安装的 Node.js 的版本不要低于本书使用的版本。

2.　NPM

NPM 是一个模块管理工具，用来管理模块的发布、下载及模块之间的依赖关系。开发 React 应用时，需要依赖很多其他的模块，这些模块就可以通过 NPM 下载。NPM 已经集成到 Node.js 的安装包中，所以不需要单独安装。另外，Facebook 联合 Exponent、Google 和 Tilde 共同推出了另一个模块管理工具 Yarn（https://yarnpkg.com），可以作为 NPM 的替代工具。本书使用的是 NPM。

1.3.2　辅助工具

1. Webpack

Webpack 是用于现代 JavaScript 应用程序的模块打包工具。Webpack 会递归地构建一个包含应用程序所需的每个模块的依赖关系图，然后将所有模块打包到少量文件中。Webpack 不仅可以打包 JS 文件，配合相关插件的使用，它还可以打包图片资源和样式文件，已经具备一站式的 JavaScript 应用打包能力。Webpack 本身就是一个模块，因此可以通过 NPM 等模块管理工具安装。

2. Babel

我们已经提到，React 应用中会大量使用 ES 6 语法，但是目前的浏览器环境并不完全支持 ES 6 语法。Babel 是一个 JavaScript 编译器，它可以将 ES 6 及其以后的语法编译成 ES 5 的语法，从而让我们可以在开发过程中尽情使用最新的 JavaScript 语法，而不需要担心代码无法在浏览器端运行的问题。Babel 一般会和 Webpack 一起使用，在 Webpack 编译打包的阶段，利用 Babel 插件将 ES 6 及其以后的语法编译成 ES 5 语法。

3. ESLint

ESLint 是一个插件化的 JavaScript 代码检测工具，它既可以用于检查常见的 JavaScript 语法错误，又可以进行代码风格检查，从而保证团队内不同开发人员编写的代码都能遵循统一的代码规范。使用 ESlint 必须要指定一套代码规范的配置，然后 ESlint 就会根据这套规范对代码进行检查。目前，业内比较好的规范是 Airbnb 的规范，但这套规范过于严格，并不一定适合所有团队。在实际使用时，可以先继承这套规范，然后在它的基础上根据实际需求对规范进行修改。

4. 代码编辑器

你可以在任何编辑器上编写 React 应用的代码，但一款好的编辑器能大大提高你的开发效率。本书推荐使用微软出品的 Visual Studio Code（简称 VS Code），它的操作简洁、功能强大，本书中的示例代码均在 VS Code 中完成，读者可自行到官方网站下载安装，地址为：https://code.visualstudio.com。此外，Sublime Text、Atom 和 WebStorm 也是使用较多的代码编辑器。

1.3.3 Create React App

Webpack、Babel 等工具是开发 React 应用所必需的，但这些工具的使用方法又比较烦琐，尤其是 Webpack 的使用，需要大量篇幅才能介绍清楚。为了避免读者还没有开始使用 React，就被各种辅助工具的使用搞得“头晕目眩”，本书借助 React 官方提供的脚手架工程 Create React App（https://github.com/facebookincubator/create-react-app）创建 React 应用。Create React App 基于最佳实践，将 Webpack、Babel、ESlint 等工具的配置做了封装，使用 Create React App 创建的项目无须进行任何配置工作，从而使开发者可以专注于应用开发。


注 意

虽然本书没有详细介绍 Webpack、Babel 等工具，但并不说明它们不重要，事实上，它们是现代 Web 开发工程化体系中的重要内容。建议读者在掌握 React 后，系统地学习这些工具。

1. 安装

打开命令行终端，依次输入以下命令：

```
npm install -g create-react-app
```

通过使用 –g 参数，我们将 create-react-app 安装到了系统的全局环境，这样就可以在任意路径下使用它了。

2. 创建应用

使用 create-react-app 创建一个新应用，在命令行终端执行：

```
create-react-app my-app
```

这时会在当前路径下新建一个名为 my-app 的文件夹，my-app 也就是我们新创建的 React 应用。

3. 运行应用

在命令行终端执行：

```
cd my-app
npm start
```

当应用启动成功后，在浏览器地址栏输入 http://localhost:3000 即可访问应用，界面如图 1-1 所示。

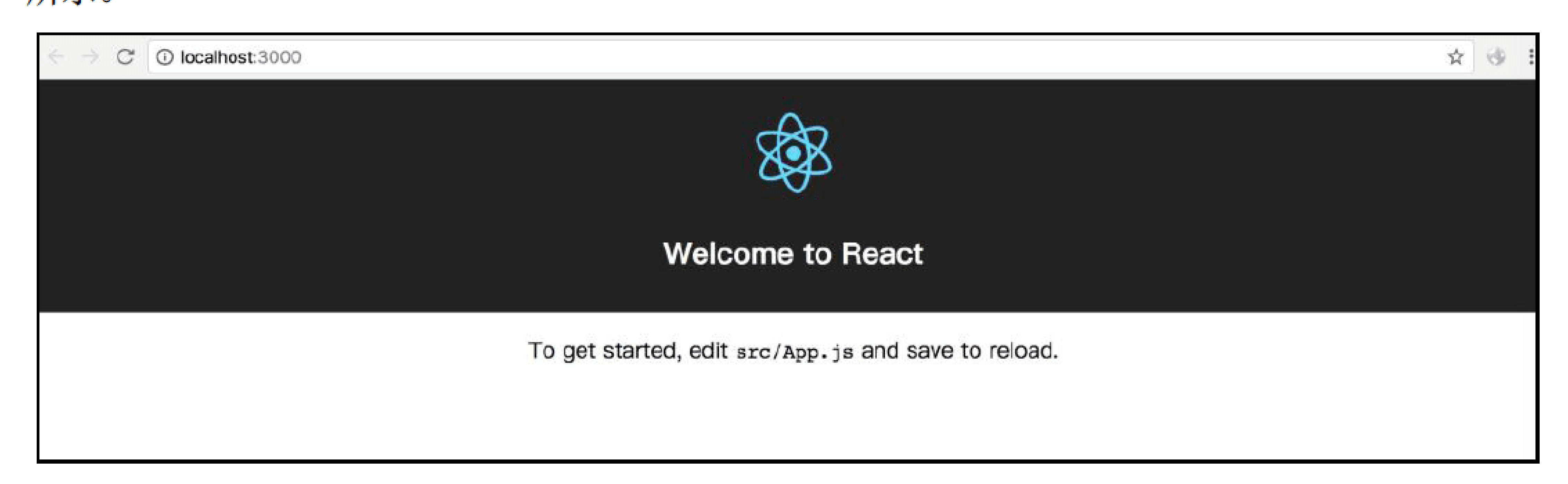
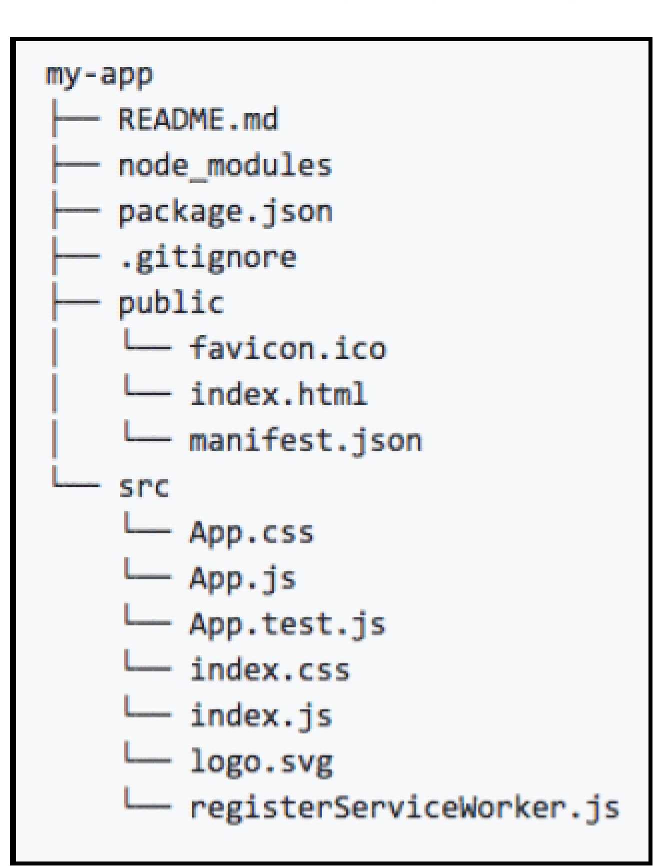


图 1-1

用 VS Code 打开 my-app 文件夹，文件夹内的目录结构如图 1-2 所示。



```
my-app
├── README.md
├── node_modules
├── package.json
├── .gitignore
├── public
│   └── favicon.ico
│   └── index.html
│   └── manifest.json
└── src
    └── App.css
    └── App.js
    └── App.test.js
    └── index.css
    └── index.js
    └── logo.svg
    └── registerServiceWorker.js
```

图 1-2

node_modules 文件夹内是安装的所有依赖模块；package.json 文件定义了项目的基本信息，如项目名称、版本号、在该项目下可执行的命令以及项目的依赖模块等；public 文件夹下的 index.html 是应用的入口页面；src 文件夹下是项目源代码，其中 index.js 是代码入口。

index.js 导入了模块 App.js，修改 App.js，将它的 render 方法修改如下：

```
render() {
  return (
    <div className="App">
      <div className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <h1>Welcome to React</h1>
      </div>
      <p className="App-intro">
        Hello, world!
      </p>
    </div>
  );
}
```

保存文件后，可以发现浏览器页面实时进行了更新，这是因为 Create React App 也包含热加载功能，可实时更新代码变化。新的页面白色背景区域显示“Hello，world!”，如图 1-3 所示。

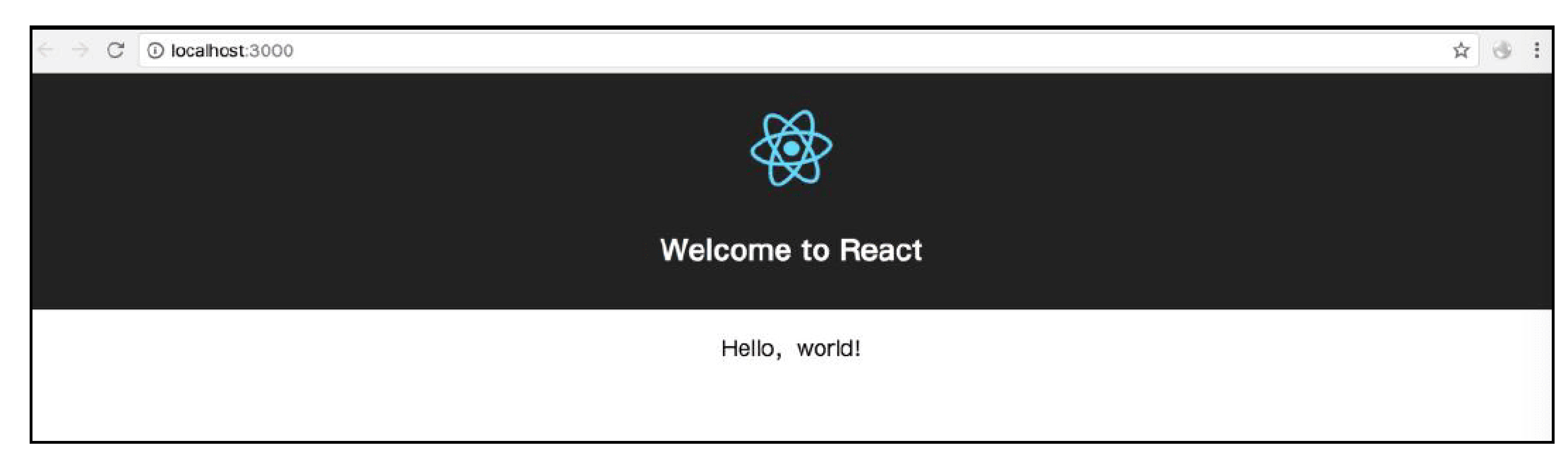


图 1-3

至此，我们完成了第一个 React 应用。

1.4　本章小结

本章介绍了 React 的基本理念和主要特性，让读者对 React 先有一个感性的认识。同时还介绍了 React 开发中常用的 ES 6 语法、React 的相关开发环境和工具，这些都是为后面讲解 React 作铺垫。从下一章开始，我们将正式介绍 React 的知识。

第 2 章

React 基础

本章将对 React 的基础知识做详细介绍，主要包括：

- JSX 语法
- 组件的概念及使用
- 列表渲染
- 事件处理
- 表单

通过学习本章内容，可以掌握 React 以组件为核心的基本开发方法。本章还会将每一小节的知识点逐步应用到一个简易 BBS 的项目实例上，让大家有更深的理解和体会。

2.1 JSX

2.1.1 JSX 简介

JSX 是一种用于描述 UI 的 JavaScript 扩展语法，React 使用这种语法描述组件的 UI。

长期以来，UI 和数据分离一直是前端领域的一个重要关注点。为了解决这个问题，前端领域发明了模板，将 UI 的定义放入模板文件，将数据的逻辑维护在 JS 代码中，然后通过模板引擎，根据数据和模板文件渲染出最终的 HTML 文件或代码片段。大部分前端框架都实现了自己的模板，要使用这些模板，必须学习它们各自的模板语法，而且对于复杂的 UI，模板语法也很难对其进行清晰的描述。

React 致力于通过组件的概念将页面进行拆分并实现组件复用。React 认为，一个组件应该是具备 UI 描述和 UI 数据的完整体，不应该将它们分开处理，于是发明了 JSX，作为 UI 描述和 UI 数据之间的桥梁。这样，在组件内部可以使用类似 HTML 的标签描述组件的 UI，让 UI 结构直观清晰，同时因为 JSX 本质上仍然是 JavaScript，所以可以使用更多的 JS 语法，构建更加复杂的 UI 结构。

2.1.2　JSX 语法

1. 基本语法

JSX 的基本语法和 XML 语法相同，都是使用成对的标签构成一个树状结构的数据，例如：

```
const element = (
  <div>
    <h1>Hello, world!</h1>
  </div>
)
```

2. 标签类型

在 JSX 语法中，使用的标签类型有两种：DOM 类型的标签（div、span 等）和 React 组件类型的标签（在 2.2 节详细介绍组件的概念）。当使用 DOM 类型的标签时，标签的首字母必须小写；当使用 React 组件类型的标签时，组件名称的首字母必须大写。React 正是通过首字母的大小写判断渲染的是一个 DOM 类型的标签还是一个 React 组件类型的标签。例如：

```
// DOM 类型标签
const element = <h1>Hello, world!</h1>;

// React 组件类型标签
const element = <HelloWorld />;

// 二者可以互相嵌套使用
const element = (
  <div>
    <HelloWorld />
  </div>;
)
```

3. JavaScript 表达式

JSX 可以使用 JavaScript 表达式，因为 JSX 本质上仍然是 JavaScript。在 JSX 中使用 JavaScript 表达式需要将表达式用大括号“{}”包起来。表达式在 JSX 中的使用场景主要有两个：通过表达式给标签属性赋值和通过表达式定义子组件。例如：

```
// 通过表达式给标签属性赋值
const element = <MyComponent foo={ 1 + 2 } />
```

```
// 通过表达式定义子组件(map 虽然是函数，但它的返回值是 JavaScript 表达式)
const todos = ['item1', 'item2', 'item3'];
const element = (
  <ul>
    {todos.map(message => <Item key={message} message={message} />)}
  </ul>
);
```

注意，JSX 中只能使用 JavaScript 表达式，而不能使用多行 JavaScript 语句。例如，下面的写法都是错误的：

```
// 错误
const element = <MyComponent foo={const val = 1 + 2; return val; } />

// 错误
let complete;
const element = (
  <div>
    {
      if(complete){
        return <CompletedList />;
      }else{
        return null;
      }
    }
  </div>
)
```

不过，JSX 中可以使用三目运算符或逻辑与（&&）运算符代替 if 语句的作用。例如：

```
// 正确
let complete;
const element = (
  <div>
    {
      complete ? <CompletedList /> : null
    }
  </div>
)

//正确
let complete;
const element = (
  <div>
    {
```

```
      complete && <CompletedList />
    }
  </div>
)
```

4. 标签属性

当 JSX 标签是 DOM 类型的标签时，对应 DOM 标签支持的属性 JSX 也支持，例如 id、class、style、onclick 等。但是，部分属性的名称会有所改变，主要的变化有：class 要写成 className，事件属性名采用驼峰格式，例如 onclick 要写成 onClick。原因是，class 是 JavaScript 的关键字，所以改成 className；React 对 DOM 标签支持的事件重新做了封装，封装时采用了更常用的驼峰命名法命名事件。例如：

```
<div id='content' className='foo' onClick={() => {console.log('Hello, React')}} />
```

当 JSX 标签是 React 组件类型时，可以任意自定义标签的属性名。例如：

```
<User name='React' age='4' address='America' >
```

5. 注释

JSX 中的注释需要用大括号“{}”将/**/包裹起来。例如：

```
const element = (
  <div>
    {/* 这里是一个注释 */}
    <span>React</span>
  </div>
)
```

2.1.3 JSX 不是必需的

JSX 语法对使用 React 来说并不是必需的，实际上，JSX 语法只是 React.createElement (component, props, ...children)的语法糖，所有的 JSX 语法最终都会被转换成对这个方法的调用。例如：

```
//JSX 语法
const element = <div className='foo'>Hello, React</div>

//转换后
const element = React.createElement('div', {className: 'foo'}, 'Hello, React')
```

虽然 JSX 只是一个语法糖，但使用它创建界面元素更加清晰简洁，在项目使用中应该首选 JSX 语法。

2.2 组　　件

2.2.1 组件定义

组件是 React 的核心概念，是 React 应用程序的基石。组件将应用的 UI 拆分成独立的、可复用的模块，React 应用程序正是由一个一个组件搭建而成的。

定义一个组件有两种方式，使用 ES 6 class（类组件）和使用函数（函数组件）。我们先介绍使用 class 定义组件的方式，使用函数定义组件的方式稍后介绍。

使用 class 定义组件需要满足两个条件：

（1）class 继承自 React.Component。

（2）class 内部必须定义 render 方法，render 方法返回代表该组件 UI 的 React 元素。

使用 create-react-app 新建一个简易 BBS 项目，在这个项目中定义一个组件 PostList，用于展示 BBS 的帖子列表。

PostList 的定义如下：

```
// PostList.js
import React, { Component } from "react";

class PostList extends Component {
  render() {
    return (
      <div>
        帖子列表：
        <ul>
          <li>大家一起来讨论 React 吧</li>
          <li>前端框架，你最爱哪一个</li>
          <li>Web App 的时代已经到来</li>
        </ul>
      </div>
    );
  }
}

export default PostList;
```

注意，在定义组件之后，使用 ES 6 export 将 PostList 作为默认模块导出，从而可以在其他 JS 文件中导入 PostList 使用。现在页面上还无法显示出 PostList 组件，因为我们还没有将 PostList 挂载到页面的 DOM 节点上。需要使用 ReactDOM.render() 完成这一个工作：

```
// index.js
import React from "react";
```

```
import ReactDOM from "react-dom";
import PostList from "./PostList";

ReactDOM.render(<PostList />, document.getElementById("root"));
```

注意，使用 ReactDOM.render() 需要先导入 react-dom 库，这个库会完成组件所代表的虚拟 DOM 节点到浏览器的 DOM 节点的转换。此时，页面展现在浏览器中，如图 2-1 所示。因为我们并没有为组件添加任何 CSS 样式，所以当前的页面效果还非常简陋，后续会逐步进行优化。本节项目源代码的目录为 /chapter-02/bbs-components。

话题列表：

- 大家一起来讨论React吧
- 前端框架，你最爱哪一个
- Web App的时代已经到来

图 2-1

2.2.2 组件的 props

在 2.2.1 小节中，PostList 中的每一个帖子都使用一个标签直接包裹，但一个帖子不仅包含帖子的标题，还会包含帖子的创建人、帖子创建时间等信息，这时候标签下的结构就会变得复杂，而且每一个帖子都需要重写一次这个复杂的结构，PostList 的结构将会变成类似这样的形式：

```
class PostList extends Component {
  render() {
    return (
      <div>
        帖子列表：
        <ul>
          <li>
              <div>大家一起来讨论 React 吧</div>
              <div>创建人：<span>张三</span></div>
              <div>创建时间：<span>2017-09-01 10：00</span></div>
          </li>
          <li>
              <div>前端框架，你最爱哪一个</div>
              <div>创建人：<span>李四</span></div>
              <div>创建时间：<span>2017-09-01 12：00</span></div>
          </li>
          <li>
              <div>Web App 的时代已经到来</div>
              <div>创建人：<span>王五</span></div>
              <div>创建时间：<span>2017-09-01 14：00</span></div>
          </li>
        </ul>
      </div>
    );
  }
}
```

这样的结构显然很冗余，我们完全可以封装一个 PostItem 组件负责每一个帖子的展示，然后在 PostList 中直接使用 PostItem 组件，这样在 PostList 中就不需要为每一个帖子重复写一堆 JSX 标签。但是，帖子列表的数据依然存在于 PostList 中，如何将数据传递给每一个 PostItem 组件呢？这时候就需要用到组件的 props 属性。组件的 props 用于把父组件中的数据或方法传递给子组件，供子组件使用。在 2.1 节中，我们介绍了 JSX 标签的属性。props 是一个简单结构的对象，它包含的属性正是由组件作为 JSX 标签使用时的属性组成。例如下面是一个使用 User 组件作为 JSX 标签的声明：

```
<User name='React' age='4' address='America' >
```

此时 User 组件的 props 结构如下：

```
props = {
  name: 'React',
  age: '4',
  address: 'America'
}
```

现在我们利用 props 定义 PostItem 组件：

```
// PostItem.js
import React, { Component } from "react";

class PostItem extends Component {
  render() {
    const { title, author, date } = this.props;
    return (
      <li>
        <div>
          {title}
        </div>
        <div>
          创建人：<span>{author}</span>
        </div>
        <div>
          创建时间：<span>{date}</span>
        </div>
      </li>
    );
  }
}

export default PostItem;
```

然后在 PostList 中使用 PostItem：

```
// PostList.js
```

```
import React, { Component } from "react";
import PostItem from "./PostItem";

// 真实项目中，帖子列表数据一般从服务器端获取
// 这里我们通过定义常量 data 存储列表数据
const data = [
  { title: "大家一起来讨论 React 吧", author: "张三", date: "2017-09-01 10:00" },
  { title: "前端框架，你最爱哪一个", author: "李四", date: "2017-09-01 12:00" },
  { title: "Web App 的时代已经到来", author: "王五", date: "2017-09-01 14:00" }
];
class PostList extends Component {
  render() {
    return (
      <div>
        帖子列表：
        <ul>
          {data.map(item =>
            <PostItem
              title={item.title}
              author={item.author}
              date={item.date}
            />
          )}
        </ul>
      </div>
    );
  }
}

export default PostList;
```

此时，页面截图如图 2-2 所示。本节项目源代码的目录为/chapter-02/bbs-components-props。

话题列表：

- 大家一起来讨论React吧
  创建人：张三
  创建时间：2017-09-01 10:00
- 前端框架，你最爱哪一个
  创建人：李四
  创建时间：2017-09-01 12:00
- Web App的时代已经到来
  创建人：王五
  创建时间：2017-09-01 14:00

图 2-2

2.2.3　组件的 state

组件的 state 是组件内部的状态，state 的变化最终将反映到组件 UI 的变化上。我们在组件的构造方法 constructor 中通过 this.state 定义组件的初始状态，并通过调用 this.setState 方法改变组件状态（也是改变组件状态的唯一方式），进而组件 UI 也会随之重新渲染。

下面来改造一下 BBS 项目。我们为每一个帖子增加一个“点赞”按钮，每点击一次，该帖子的点赞数增加 1。点赞数是会发生变化的，它的变化也会影响到组件 UI，因此我们将点赞数 vote 作为 PostItem 的一个状态定义到它的 state 内。

```
import React, { Component } from "react";

class PostItem extends Component {
  constructor(props) {
    super(props);
    this.state = {
      vote: 0
    };
  }
  // 处理点赞逻辑
  handleClick() {
    let vote = this.state.vote;
    vote++;
    this.setState({
      vote: vote
    });
  }

  render() {
    const { title, author, date } = this.props;
    return (
      <li>
        <div>
          {title}
        </div>
        <div>
          创建人：<span>{author}</span>
        </div>
        <div>
          创建时间：<span>{date}</span>
        </div>
        <div>
          <button
            onClick={() => {
              this.handleClick();
```

```
            }}
          >
            点赞
          </button>
           
          <span>
            {this.state.vote}
          </span>
        </div>
      </li>
    );
  }
}

export default PostItem;
```

这里有三个需要注意的地方：

（1）在组件的构造方法 constructor 内，首先要调用 super(props)，这一步实际上是调用了 React.Component 这个 class 的 constructor 方法，用来完成 React 组件的初始化工作。

（2）在 constructor 中，通过 this.state 定义了组件的状态。

（3）在 render 方法中，我们为标签定义了处理点击事件的响应函数，在响应函数内部会调用 this.setState 更新组件的点赞数。

新页面的截图如图 2-3 所示。本节项目源代码的目录为/chapter-02/bbs-components-state。

通过 2.2.2 和 2.2.3 两个小节的介绍可以发现，组件的 props 和 state 都会直接影响组件的 UI。事实上，React 组件可以看作一个函数，函数的输入是 props 和 state，函数的输出是组件的 UI。

```
UI = Component(props,state)
```

话题列表：

- 大家一起来讨论React吧
  创建人：张三
  创建时间：2017-09-01 10:00
  点赞 1
- 前端框架，你最爱哪一个
  创建人：李四
  创建时间：2017-09-01 12:00
  点赞 2
- Web App的时代已经到来
  创建人：王五
  创建时间：2017-09-01 14:00
  点赞 0

图 2-3

React 组件正是由 props 和 state 两种类型的数据驱动渲染出组件 UI。props 是组件对外的接口，组件通过 props 接收外部传入的数据（包括方法）；state 是组件对内的接口，组件内部状态的变化通过 state 来反映。另外，props 是只读的，你不能在组件内部修改 props；state 是可变的，组件状态的变化通过修改 state 来实现。在第 4 章中，我们还会对 props 和 state 进行详细比较。

2.2.4　有状态组件和无状态组件

是不是每个组件内部都需要定义 state 呢？当然不是。state 用来反映组件内部状态的变化，如果一个组件的内部状态是不变的，当然就用不到 state，这样的组件称之为无状态组件，例如 PostList。反之，一个组件的内部状态会发生变化，就需要使用 state 来保存变化，这样的组件称之为有状态组件，例如 PostItem。

定义无状态组件除了使用 ES 6 class 的方式外，还可以使用函数定义，也就是我们在本节开始时所说的函数组件。一个函数组件接收 props 作为参数，返回代表这个组件 UI 的 React 元素结构。例如，下面是一个简单的函数组件：

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

可以看出，函数组件的写法比类组件的写法要简洁很多，在使用无状态组件时，应该尽量将其定义成函数组件。

在开发 React 应用时，一定要先认真思考哪些组件应该设计成有状态组件，哪些组件应该设计成无状态组件。并且，应该尽可能多地使用无状态组件，无状态组件不用关心状态的变化，只聚焦于 UI 的展示，因而更容易被复用。React 应用组件设计的一般思路是，通过定义少数的有状态组件管理整个应用的状态变化，并且将状态通过 props 传递给其余的无状态组件，由无状态组件完成页面绝大部分 UI 的渲染工作。总之，有状态组件主要关注处理状态变化的业务逻辑，无状态组件主要关注组件 UI 的渲染。

下面让我们回过头来看一下 BBS 项目的组件设计。当前的组件设计并不合适，主要体现在：

（1）帖子列表通过一个常量 data 保存在组件之外，但帖子列表的数据是会改变的，新帖子的增加或原有帖子的删除都会导致帖子列表数据的变化。

（2）每一个 PostItem 都维持一个 vote 状态，但除了 vote 以外，帖子其他的信息（如标题、创建人等）都保存在 PostList 中，这显然也是不合理的。

我们对这两个组件进行重新设计，将 PostList 设计为有状态组件，负责帖子列表数据的获取以及点赞行为的处理，将 PostItem 设计为无状态组件，只负责每一个帖子的展示。此时，PostList 和 PostItem 重构如下：

```
// PostList.js
import React, { Component } from "react";
import PostItem from "./PostItem";

class PostList extends Component {
  constructor(props) {
```

```
      super(props);
      this.state = {
        posts: []
      };
      this.timer = null;   // 定时器
      this.handleVote = this.handleVote.bind(this);  //ES 6 class 中，必须手动绑
定方法 this 的指向
    }

    componentDidMount() {
      // 用 setTimeout 模拟异步从服务器端获取数据
      this.timer = setTimeout(() => {
        this.setState({
          posts: [
            { id: 1, title: "大家一起来讨论 React 吧", author: "张三", date:
"2017-09-01 10:00", vote: 0 },
            { id: 2, title: "前端框架，你最爱哪一个", author: "李四", date: "2017-09-01
12:00", vote: 0 },
            { id: 3, title: "Web App 的时代已经到来", author: "王五", date:
"2017-09-01 14:00", vote: 0 }
          ]
        });
      }, 1000);
    }

    componentWillUnmount() {
      if(this.timer) {
        clearTimeout(this.timer);    // 清除定时器
      }
    }

    handleVote(id) {
       //根据帖子 id 进行过滤，找到待修改 vote 属性的帖子，返回新的 posts 对象
      const posts = this.state.posts.map(item => {
         const newItem = item.id === id ? {...item, vote: ++item.vote} : item;
        return newItem;
      });
      // 使用新的 posts 对象设置 state
      this.setState({
        posts
      });
    }
```

```
  render() {
    return (
      <div>
        帖子列表：
        <ul>
          {this.state.posts.map(item =>
            <PostItem
              post = {item}
              onVote = {this.handleVote}
            />
          )}
        </ul>
      </div>
    );
  }
}

export default PostList;

// PostItem.js
import React from "react";

function PostItem(props) {
  const handleClick = () => {
    props.onVote(props.post.id);
  };
  const { post } = props;
  return (
    <li>
      <div>
        {post.title}
      </div>
      <div>
        创建人：<span>{post.author}</span>
      </div>
      <div>
        创建时间：<span>{post.date}</span>
      </div>
      <div>
        <button onClick={handleClick}>点赞</button>
         
        <span>{post.vote}</span>
      </div>
```

```
    </li>
  );
}

export default PostItem;
```

这里主要的修改有：

（1）帖子列表数据定义为 PostList 组件的一个状态。

（2）在 componentDidMount 生命周期方法中（关于组件的生命周期将在 2.3 节详细介绍）通过 setTimeout 设置一个延时，模拟从服务器端获取数据，然后调用 setState 更新组件状态。

（3）将帖子的多个属性（ID、标题、创建人、创建时间、点赞数）合并成一个 post 对象，通过 props 传递给 PostItem。

（4）在 PostList 内定义 handleVote 方法，处理点赞逻辑，并将该方法通过 props 传递给 PostItem。

（5）PostItem 定义为一个函数组件，根据 PostList 传递的 post 属性渲染 UI。当发生点赞行为时，调用 props.onVote 方法将点赞逻辑交给 PostList 中的 handleVote 方法处理。

这样修改后，PostItem 只关注如何展示帖子，至于帖子的数据从何而来以及点赞逻辑如何处理，统统交给有状态组件 PostList 处理。组件之间解耦更加彻底，PostItem 组件更容易被复用。本节项目源代码的目录为/chapter-02/bbs-components-stateless。

2.2.5　属性校验和默认属性

我们已经知道，props 是一个组件对外暴露的接口，但到目前为止，组件内部并没有明显地声明它暴露出哪些接口，以及这些接口的类型是什么，这不利于组件的复用。幸运的是，React 提供了 PropTypes 这个对象，用于校验组件属性的类型。PropTypes 包含组件属性所有可能的类型，我们通过定义一个对象（对象的 key 是组件的属性名，value 是对应属性的类型）实现组件属性类型的校验。例如：

```
import PropTypes from 'prop-types';

class PostItem extends React.Component {
  //......
}

PostItem.propTypes = {
  post: PropTypes.object,
  onVote: PropTypes.func
};
```

PropTypes 可以校验的组件属性类型见表 2-1。

表2-1　组件属性类型和PropTypes属性的对应关系

类型	PropTypes 对应属性
String	PropTypes.string
Number	PropTypes.number
Boolean	PropTypes.bool
Function	PropTypes.func
Object	PropTypes.object
Array	PropTypes.array
Symbol	PropTypes.symbol
Element（React 元素）	PropTypes.element
Node（可被渲染的节点：数字、字符串、React 元素或由这些类型的数据组成的数组）	PropTypes.node

当使用 PropTypes.object 或 PropTypes.array 校验属性类型时，我们只知道这个属性是一个对象或一个数组，至于对象的结构或数组元素的类型是什么样的，依然无从得知。这种情况下，更好的做法是使用 PropTypes.shape 或 PropTypes.arrayOf。例如：

```
style: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),
sequence: PropTypes.arrayOf(PropTypes.number)
```

表示 style 是一个对象，对象有 color 和 fontSize 两个属性，color 是字符串类型，fontSize 是数字类型；sequence 是一个数组，数组的元素是数字。

如果属性是组件的必需属性，也就是当使用某个组件时，必须传入的属性，就需要在 PropTypes 的类型属性上调用 isRequired。在 BBS 项目中，对于 PostItem 组件，post 和 onVote 都是必需属性，PostItem 的 propTypes 定义如下：

```
PostItem.propTypes = {
  post: PropTypes.shape({
    id: PropTypes.number,
    title: PropTypes.string,
    author: PropTypes.string,
    date: PropTypes.string,
    vote: PropTypes.number
  }).isRequired,
  onVote: PropTypes.func.isRequired
}
```

本节项目源代码的目录为/chapter-02/bbs-components-propTypes。

React 还提供了为组件属性指定默认值的特性，这个特性通过组件的 defaultProps 实现。当组件属性未被赋值时，组件会使用 defaultProps 定义的默认属性。例如：

```
function Welcome(props) {
  return <h1 className='foo'>Hello, {props.name}</h1>;
}

Welcome.defaultProps = {
  name: 'Stranger'
};
```

2.2.6 组件样式

到目前为止，我们还未对组件添加任何样式。本节将介绍如何为组件添加样式。

为组件添加样式的方法主要有两种：外部 CSS 样式表和内联样式。

1. 外部 CSS 样式表

这种方式和我们平时开发 Web 应用时使用外部 CSS 文件相同，CSS 样式表中根据 HTML 标签类型、ID、class 等选择器定义元素的样式。唯一的区别是，React 元素要使用 className 来代替 class 作为选择器。例如，为 Welcome 组件的根节点设置一个 className='foo'的属性：

```
function Welcome(props) {
  return <h1 className='foo'>Hello, {props.name}</h1>;
}
```

然后在 CSS 样式表中通过 class 选择器定义 Welcome 组件的样式：

```
// style.css
.foo {
  width:100%;
  height:50px;
  background-color:blue;
  font-size: 20px;
}
```

样式表的引入方式有两种，一种是在使用组件的 HTML 页面中通过标签引入：

```
<link rel="stylesheet" type="text/css" href="style.css">
```

另一种是把样式表文件当作一个模块，在使用该样式表的组件中，像导入其他组件一样导入样式表文件：

```
import './style.css'; //要保证相对路径设置正确

function Welcome(props) {
  return <h1 className='foo'>Hello, {props.name}</h1>;
}
```

第一种引入样式表的方式常用于该样式表文件作用于整个应用的所有组件（一般是基础样式表）；第二种引入样式表的方式常用于该样式表作用于某个组件（相当于组件的私有样式），全局

的基础样式表也可以使用第二种方式引入，一般在应用的入口 JS 文件中引入。

补充说明：使用 CSS 样式表经常遇到的一个问题是 class 名称冲突。业内解决这个问题的一个常用方案是使用 CSS Modules，CSS Modules 会对样式文件中的 class 名称进行重命名从而保证其唯一性，但 CSS Modules 并不是必需的，create-react-app 创建的项目，默认配置也是不支持这一特性的。CSS Modules 的使用并不复杂，感兴趣的读者可自行了解（参考地址：https://github.com/css-modules/css-modules）。

2. 内联样式

内联样式实际上是一种 CSS in JS 的写法：将 CSS 样式写到 JS 文件中，用 JS 对象表示 CSS 样式，然后通过 DOM 类型节点的 style 属性引用相应样式对象。依然使用 Welcome 组件举例：

```
function Welcome(props) {
  return (
    <h1
      style={{
        width: "100%",
        height: "50px",
        backgroundColor: "blue",
        fontSize: "20px"
      }}
    >
      Hello, {props.name}
    </h1>
  );
}
```

style 使用了两个大括号，这可能会让你感到迷惑。其实，第一个大括号表示 style 的值是一个 JavaScript 表达式，第二个大括号表示这个 JavaScript 表达式是一个对象。换一种写法就容易理解了：

```
function Welcome(props) {
  const style = {
    width: "100%",
    height: "50px",
    backgroundColor: "blue",
    fontSize: "20px"
  };
  return <h1 style={style}>Hello, {props.name}</h1>;
}
```

当使用内联样式时，还有一点需要格外注意：样式的属性名必须使用驼峰格式的命名。所以，在 Welcome 组件中，background-color 写成 backgroundColor，font-size 写成 fontSize。

下面为 BBS 项目增加一些样式。创建 style.css、PostList.css 和 PostItem.css 三个样式文件，三个样式表分别在 index.html、PostList.js、PostItem.js 中引入。样式文件如下：

```
// style.css
body {
  margin: 0;
  padding: 0;
  font-family: sans-serif;
}

ul {
  list-style: none;
}

h2 {
  text-align: center;
}

// PostList.css
.container {
  width: 900px;
  margin: 20px auto;
}

// PostItem.css
.item {
  border-top: 1px solid grey;
  padding: 15px;
  font-size: 14px;
  color: grey;
  line-height: 21px;
}

.title {
  font-size: 16px;
  font-weight: bold;
  line-height: 24px;
  color: #009a61;
}

.like {
  width: 100%;
  height: 20px;
}
```

```
.like img{
  width: 20px;
  height: 20px;
}

.like span{
  width: 20px;
  height: 20px;
  vertical-align: middle;
  display: table-cell;
}
```

这里需要提醒一下，style.css 放置在 public 文件夹下，PostList.css 和 PostItem.css 放置在 src 文件夹下。create-react-app 将 public 下的文件配置成可以在 HTML 页面中直接引用，因此我们将 style.css 放置在 public 文件夹下。而 PostList.css 和 PostItem.css 是以模块的方式在 JS 文件中被导入的，因此放置在 src 文件夹下。

我们还将 PostItem 中的点赞按钮换成了图标，图标也可以作为一个模块被 JS 文件导入，如 PostItem.js 所示：

```
import React from "react";
import "./PostItem.css";
import like from "./images/like-default.png";   //图标作为模块被导入

function PostItem(props) {
  const handleClick = () => {
    props.onVote(props.post.id);
  };
  const { post } = props;
  return (
    <li className='item'>
      <div className='title'>
        {post.title}
      </div>
      <div>
        创建人：<span>{post.author}</span>
      </div>
      <div>
        创建时间：<span>{post.date}</span>
      </div>
      <div className='like'>
         <span><img src={like} onClick={handleClick} /></span>
         <span>{post.vote}</span>
      </div>
```

```
    </li>
  );
}

export default PostItem;
```

增加样式后的页面截图如图 2-4 所示。本节项目源代码的目录为/chapter-02/bbs-components-style。



图 2-4

2.2.7 组件和元素

在 2.1 节介绍过 React 元素的概念。React 组件和元素这两个概念非常容易混淆。React 元素是一个普通的 JavaScript 对象，这个对象通过 DOM 节点或 React 组件描述界面是什么样子的。JSX 语法就是用来创建 React 元素的（不要忘了，JSX 语法实际上是调用了 React.createElement 方法）。例如：

```
//Button 是一个自定义的 React 组件
<div className='foo'>
  <Button color='blue'>
    OK
  </Button>
</div>
```

上面的 JSX 代码会创建下面的 React 元素：

```
{
  type: 'div',
  props: {
    className: 'foo',
    children: {
      type: 'Button',
      props: {
```

```
        color: 'blue',
        children: 'OK'
      }
    }
  }
}
```

React 组件是一个 class 或函数，它接收一些属性作为输入，返回一个 React 元素。React 组件是由若干 React 元素组建而成的。通过下面的例子，可以解释 React 组件与 React 元素间的关系。

```
// Button 是一个 React 组件
class Button extends React.Component {
  render() {
    return (<button>OK</button>);
  }
}

// 在 JSX 中使用组件 Button， button 是一个代表组件 Button 的 React 元素
const button = <Button />;

// 在组件 Page 中使用 React 元素 button
class Page extends React.Component {
  render() {
    return (
      <div>
         {button}
      </div>
    );
  }
}

// 上面的 Page 写法等价于下面这种写法：
class Page extends React.Component {
  render() {
    return (
      <div>
         <Button />
      </div>
    );
  }
}
```

2.3 组件的生命周期

组件从被创建到被销毁的过程称为组件的生命周期。React 为组件在不同的生命周期阶段提供不同的生命周期方法，让开发者可以在组件的生命周期过程中更好地控制组件的行为。通常，组件的生命周期可以被分为三个阶段：挂载阶段、更新阶段、卸载阶段。

2.3.1 挂载阶段

这个阶段组件被创建，执行初始化，并被挂载到 DOM 中，完成组件的第一次渲染。依次调用的生命周期方法有：

（1）constructor

（2）componentWillMount

（3）render

（4）componentDidMount

1. constructor

这是 ES 6 class 的构造方法，组件被创建时，会首先调用组件的构造方法。这个构造方法接收一个 props 参数，props 是从父组件中传入的属性对象，如果父组件中没有传入属性而组件自身定义了默认属性，那么这个 props 指向的就是组件的默认属性。你必须在这个方法中首先调用 super(props)才能保证 props 被传入组件中。constructor 通常用于初始化组件的 state 以及绑定事件处理方法等工作。

2. componentWillMount

这个方法在组件被挂载到 DOM 前调用，且只会被调用一次。这个方法在实际项目中很少会用到，因为可以在该方法中执行的工作都可以提前到 constructor 中。在这个方法中调用 this.setState 不会引起组件的重新渲染。

3. render

这是定义组件时唯一必要的方法（组件的其他生命周期方法都可以省略）。在这个方法中，根据组件的 props 和 state 返回一个 React 元素，用于描述组件的 UI，通常 React 元素使用 JSX 语法定义。需要注意的是，render 并不负责组件的实际渲染工作，它只是返回一个 UI 的描述，真正的渲染出页面 DOM 的工作由 React 自身负责。render 是一个纯函数，在这个方法中不能执行任何有副作用的操作，所以不能在 render 中调用 this.setState，这会改变组件的状态。

4. componentDidMount

在组件被挂载到 DOM 后调用，且只会被调用一次。这时候已经可以获取到 DOM 结构，因此依赖 DOM 节点的操作可以放到这个方法中。这个方法通常还会用于向服务器端请求数据。在这个方法中调用 this.setState 会引起组件的重新渲染。

2.3.2　更新阶段

组件被挂载到 DOM 后，组件的 props 或 state 可以引起组件更新。props 引起的组件更新，本质上是由渲染该组件的父组件引起的，也就是当父组件的 render 方法被调用时，组件会发生更新过程，这个时候，组件 props 的值可能发生改变，也可能没有改变，因为父组件可以使用相同的对象或值为组件的 props 赋值。但是，无论 props 是否改变，父组件 render 方法每一次调用，都会导致组件更新。State 引起的组件更新，是通过调用 this.setState 修改组件 state 来触发的。组件更新阶段，依次调用的生命周期方法有：

（1）componentWillReceiveProps
（2）shouldComponentUpdate
（3）componentWillUpdate
（4）render
（5）componentDidUpdate

1. componentWillReceiveProps（nextProps）

这个方法只在 props 引起的组件更新过程中，才会被调用。State 引起的组件更新并不会触发该方法的执行。方法的参数 nextProps 是父组件传递给当前组件的新的 props。但如上文所述，父组件 render 方法的调用并不能保证传递给子组件的 props 发生变化，也就是说 nextProps 的值可能和子组件当前 props 的值相等，因此往往需要比较 nextProps 和 this.props 来决定是否执行 props 发生变化后的逻辑，比如根据新的 props 调用 this.setState 触发组件的重新渲染。



（1）在 componentWillReceiveProps 中调用 setState，只有在组件 render 及其之后的方法中，this.state 指向的才是更新后的 state。在 render 之前的方法 shouldComponentUpdate、componentWillUpdate 中，this.state 依然指向的是更新前的 state。

（2）通过调用 setState 更新组件状态并不会触发 componentWillReceiveProps 的调用，否则可能会进入一个死循环，componentWillReceiveProps → this.setState → componentWillReceiveProps→this.setState……

2. shouldComponentUpdate（nextProps, nextState）

这个方法决定组件是否继续执行更新过程。当方法返回 true 时（true 也是这个方法的默认返回值），组件会继续更新过程；当方法返回 false 时，组件的更新过程停止，后续的 componentWillUpdate、render、componentDidUpdate 也不会再被调用。一般通过比较 nextProps、nextState 和组件当前的 props、state 决定这个方法的返回结果。这个方法可以用来减少组件不必要的渲染，从而优化组件的性能。

3. componentWillUpdate（nextProps, nextState）

这个方法在组件 render 调用前执行，可以作为组件更新发生前执行某些工作的地方，一般也很少用到。


注 意

shouldComponentUpdate 和 componentWillUpdate 中都不能调用 setState，否则会引起循环调用问题，render 永远无法被调用，组件也无法正常渲染。

4. componentDidUpdate（prevProps, prevState）

组件更新后被调用，可以作为操作更新后的 DOM 的地方。这个方法的两个参数 prevProps、prevState 代表组件更新前的 props 和 state。

2.3.3 卸载阶段

组件从 DOM 中被卸载的过程，这个过程中只有一个生命周期方法：

```
componentWillUnmount
```

这个方法在组件被卸载前调用，可以在这里执行一些清理工作，比如清除组件中使用的定时器，清除 componentDidMount 中手动创建的 DOM 元素等，以避免引起内存泄漏。

最后还需要提醒大家，只有类组件才具有生命周期方法，函数组件是没有生命周期方法的，因此永远不要在函数组件中使用生命周期方法。

2.4 列表和 Keys

在组件中渲染列表数据是非常常见的场景，例如，BBS 项目 PostList 组件就需要根据列表数据 posts 进行渲染：

```
class PostList extends Component {

  /** 省略其余代码 **/

  render() {
    return (
      <div className='container'>
        <h2>帖子列表</h2>
        <ul>
          {this.state.posts.map(item =>
            <PostItem
              post = {item}
              onVote = {this.handleVote}
            />
          )}
        </ul>
      </div>
    );
  }
}
```

下面运行 BBS 项目，然后打开 Chrome 浏览器的控制台，可以看到如图 2-5 所示的警告信息。

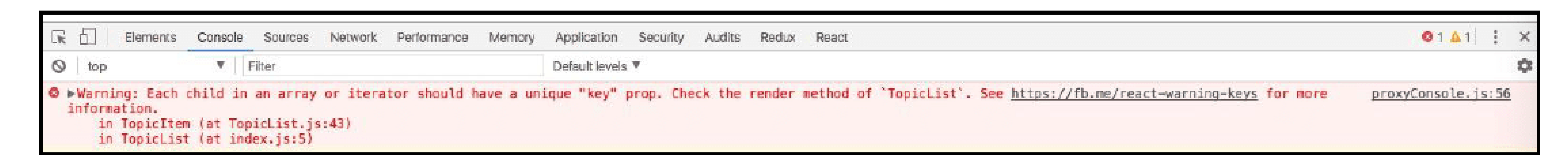


图 2-5

警告信息提示我们，应该为列表中的每个元素添加一个名为 key 的属性。那么这个属性有什么作用呢？原来，React 使用 key 属性来标记列表中的每个元素，当列表数据发生变化时，React 就可以通过 key 知道哪些元素发生了变化，从而只重新渲染发生变化的元素，提高渲染效率。

一般使用列表数据的 ID 作为 key 值，例如可以使用帖子的 ID 作为每一个 PostItem 的 key：

```
class PostList extends Component {

  /** 省略其余代码 **/

  render() {
    return (
      <div className='container'>
        <h2>帖子列表</h2>
        <ul>
          {this.state.posts.map(item =>
         {/* 将 id 赋值给 key 属性，作为唯一标识 */}
            <PostItem
              key = {item.id}
              post = {item}
              onVote = {this.handleVote}
            />
          )}
        </ul>
      </div>
    );
  }
}
```

再次运行程序，你会发现之前的警告消息已经不存在了。本节项目源代码的目录为/chapter-02/bbs-components-keys。

如果列表包含的元素没有 ID，也可以使用元素在列表中的位置索引作为 key 值，例如：

```
// 省略其他
{this.state.posts.map((item,index) =>
  <PostItem
     key = {index}
     post = {item}
```

```
      onVote = {this.handleVote}
    />
  )}
```

但并不推荐使用索引作为 key，因为一旦列表中的数据发生重排，数据的索引也会发生变化，不利于 React 的渲染优化。我们还会在第 5 章中详细说明这一情况。

虽然列表元素的 key 不能重复，但这个唯一性仅限于在当前列表中，而不是全局唯一。例如在一个组件中两次使用 post.id 作为列表数据的 key：

```
function Blog(props) {
  // 侧边栏导航区
  const sidebar = (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
  // 帖子列表
  const content = props.posts.map((post) =>
    <div key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
    </div>
  );
  return (
    <div>
      {sidebar}
      {content}
    </div>
  );
}

// posts 结构
const posts = [
  {id: 1, title: 'Hello React', content: 'Welcome to learning React!'},
  {id: 2, title: 'Installation', content: 'You can install React from npm.'}
];
```

2.5　事件处理

在 BBS 应用中的点赞功能已经涉及 React 中的事件处理，本节将详细介绍如何在 React 中处理事件。在 React 元素中绑定事件有两点需要注意：

（1）在 React 中，事件的命名采用驼峰命名方式，而不是 DOM 元素中的小写字母命名方式。例如，onclick 要写成 onClick，onchange 要写成 onChange 等。

（2）处理事件的响应函数要以对象的形式赋值给事件属性，而不是 DOM 中的字符串形式。例如，在 DOM 中绑定一个点击事件这样写：

```
<button onclick="clickButton()">
  Click
</button>
```

而在 React 元素中绑定一个点击事件变成这种形式：

```
<button onclick={clickButton}>   //clickButton 是一个函数
  Click
</button>
```

React 中的事件是合成事件，并不是原生的 DOM 事件。React 根据 W3C 规范定义了一套兼容各个浏览器的事件对象。在 DOM 事件中，可以通过处理函数返回 false 来阻止事件的默认行为，但在 React 事件中，必须显式地调用事件对象的 preventDefault 方法来阻止事件的默认行为。除了这一点外，DOM 事件和 React 事件在使用上并无差别。如果在某些场景下必须使用 DOM 提供的原生事件，可以通过 React 事件对象的 nativeEvent 属性获取。

其实，在 React 组件中处理事件最容易出错的地方是事件处理函数中 this 的指向问题，因为 ES 6 class 并不会为方法自动绑定 this 到当前对象。React 事件处理函数的写法主要有三种方式，不同的写法解决 this 指向问题的方式也不同。

1. 使用箭头函数

直接在 React 元素中采用箭头函数定义事件的处理函数，例如：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {number: 0};
  }

  render() {
    return (
      <button onClick={(event)=>{console.log(this.state.number);}}>
        Click
```

```
      </button>
    );
  }
}
```

因为箭头函数中的 this 指向的是函数定义时的对象，所以可以保证 this 总是指向当前组件的实例对象。当事件处理逻辑比较复杂时，如果把所有的逻辑直接写在 onClick 的大括号内，就会导致 render 函数变得臃肿，不容易直观地看出组件的 UI 结构，代码可读性也不好。这时，可以把逻辑封装成组件的一个方法，然后在箭头函数中调用这个方法。代码如下：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {number: 0};
  }
  // 每点击一次 Button，state 中的 number 增加 1
  handleClick(event) {
    const number = ++this.state.number;
    this.setState({
      number: number
    });
  }

  render() {
    return (
      <div>
        <div>{this.state.number}</div>
        <button onClick={(event)=>{this.handleClick(event);}}>
          Click
        </button>
      </div>
    );
  }
}
```

直接在 render 方法中为元素事件定义事件处理函数，最大的问题是，每次 render 调用时，都会重新创建一个新的事件处理函数，带来额外的性能开销，组件所处层级越低，这种开销就越大，因为任何一个上层组件的变化都可能会触发这个组件的 render 方法。当然，在大多数情况下，这点性能损失是可以不必在意的。

2. 使用组件方法

直接将组件的方法赋值给元素的事件属性，同时在类的构造函数中，将这个方法的 this 绑定到当前对象。例如：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {number: 0};
    this.handleClick = this.handleClick.bind(this);
  }
  // 每点击一次 Button，state 中的 number 增加 1
  handleClick(event) {
    const number = ++this.state.number;
    this.setState({
      number: number
    });
  }

  render() {
    return (
      <div>
        <div>{this.state.number}</div>
        <button onClick={this.handleClick}>
          Click
        </button>
      </div>
    );
  }
}
```

这种方式的好处是每次 render 不会重新创建一个回调函数，没有额外的性能损失。但在构造函数中，为事件处理函数绑定 this，尤其是存在多个事件处理函数需要绑定时，这种模板式的代码还是会显得烦琐。

有些开发者还习惯在为元素的事件属性赋值时，同时为事件处理函数绑定 this，例如：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {number: 0};
  }

  // 每点击一次 Button，state 中的 number 增加 1
  handleClick(event) {
    const number = ++this.state.number;
    this.setState({
      number: number
    });
  }
```

```
  render() {
    return (
      <div>
        <div>{this.state.number}</div>
        {/* 事件属性赋值和 this 绑定同时*/}
        <button onClick={this.handleClick.bind(this)}>
          Click
        </button>
      </div>
    );
  }
}
```

使用 bind 会创建一个新的函数，因此这种写法依然存在每次 render 都会创建一个新函数的问题。但在需要为处理函数传递额外参数时，这种写法就有了用武之地。例如，下面的例子需要为 handleClick 传入参数 item：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      list: [1,2,3,4],
      current: 1
    };
  }
  // 点击每一项时，将点击项设置为当前选中项，因此需要把点击项作为参数传递
  handleClick(item, event) {
    this.setState({
      current: item
    });
  }

  render() {
    return (
      <ul>
        {this.state.list.map(
          (item)=>(
            {/* bind 除了绑定 this，还绑定 item 作为参数，供 handleClick 使用 */}
            <li className={this.state.current === item ? 'current':''}
            onClick={this.handleClick.bind(this, item)}>{item}
            </li>
          )
        )}
```

```
      </ul>
    );
  }
}
```

3. 属性初始化语法（property initializer syntax）

使用 ES 7 的 property initializers 会自动为 class 中定义的方法绑定 this。例如：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {number: 0};
  }
  // ES7 的属性初始化语法，实际上也是使用了箭头函数
  handleClick = (event) => {
    const number = ++this.state.number;
    this.setState({
      number: number
    });
  }

  render() {
    return (
      <div>
        <div>{this.state.number}</div>
        <button onClick={this.handleClick}>
          Click
        </button>
      </div>
    );
  }
}
```

这种方式既不需要在构造函数中手动绑定 this，也不需要担心组件重复渲染导致的函数重复创建问题。但是，property initializers 这个特性还处于试验阶段，默认是不支持的。不过，使用官方脚手架 Create React App 创建的项目默认是支持这个特性的。你也可以自行在项目中引入 babel 的 transform-class-properties 插件获取这个特性支持。

2.6　表　　单

在有交互的 Web 应用中，表单是必不可少的。但是，和其他元素相比，表单元素在 React 中的工作方式存在一些不同。像 div、p、span 等非表单元素只需根据组件的属性或状态进行渲染即

可，但表单元素自身维护一些状态，而这些状态默认情况下是不受 React 控制的。例如，input 元素会根据用户的输入自动改变显示的内容，而不是从组件的状态中获取显示的内容。我们称这类状态不受 React 控制的表单元素为非受控组件。在 React 中，状态的修改必须通过组件的 state，非受控组件的行为显然有悖于这一原则。为了让表单元素状态的变更也能通过组件的 state 管理，React 采用受控组件的技术达到这一目的。

2.6.1　受控组件

如果一个表单元素的值是由 React 来管理的，那么它就是一个受控组件。React 组件渲染表单元素，并在用户和表单元素发生交互时控制表单元素的行为，从而保证组件的 state 成为界面上所有元素状态的唯一来源。对于不同的表单元素，React 的控制方式略有不同，下面我们就来看一下三类常用表单元素的控制方式。

1. 文本框

文本框包含类型为 text 的 input 元素和 textarea 元素。它们受控的主要原理是，通过表单元素的 value 属性设置表单元素的值，通过表单元素的 onChange 事件监听值的变化，并将变化同步到 React 组件的 state 中。下面是一个例子。

```
class LoginForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {name: '', password: ''};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  // 监听用户名和密码两个 input 值的变化
  handleChange(event) {
    const target = event.target;
    this.setState({[target.name]: target.value});
  }
  // 表单提交的响应函数
  handleSubmit(event) {
    console.log('login successfully');
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          用户名:
          {/* 通过 value 设置 input 显示内容，通过 onChange 监听 value 的变化 */}
```

```
            <input type="text" name="name" value={this.state.name}
onChange={this.handleChange} />
          </label>
          <label>
            密码：
            <input type="password" name="password" value={this.state.password}
onChange={this.handleChange} />
          </label>
          <input type="submit" value="登录" />
        </form>
      );
    }
  }
```

用户名和密码两个表单元素的值是从组件的 state 中获取的，当用户更改表单元素的值时，onChange 事件会被触发，对应的 handleChange 处理函数会把变化同步到组件的 state，新的 state 又会触发表单元素重新渲染，从而实现对表单元素状态的控制。

这个例子还包含一个处理多个表单元素的技巧：通过为两个 input 元素分别指定 name 属性，使用同一个函数 handleChange 处理元素值的变化，在处理函数中根据元素的 name 属性区分事件的来源。这样的写法显然比为每一个 input 元素指定一个处理函数简洁得多。

textarea 的使用方式和 input 几乎一致，这里不再赘述。

2. 列表

列表 select 元素是最复杂的表单元素，它可以用来创建一个下拉列表：

```
<select>
  <option value="react">React</option>
  <option value="redux">Redux</option>
  <option selected value="mobx">MobX</option>
</select>
```

通过指定 selected 属性可以定义哪一个选项（option）处于选中状态，所以上面的例子中，Mobx 这一选项是列表的初始值，处于选中状态。在 React 中，对 select 的处理方式有所不同，它通过在 select 上定义 value 属性来决定哪一个 option 元素处于选中状态。这样，对 select 的控制只需要在 select 这一个元素上修改即可，而不需要关注 option 元素。下面是一个例子：

```
class ReactStackForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'mobx'};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  // 监听下拉列表的变化
```

```
  handleChange(event) {
    this.setState({value: event.target.value});
  }
  // 表单提交的响应函数
  handleSubmit(event) {
    alert('You picked ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick one library:
          {/* select 的 value 属性定义当前哪个 option 元素处于选中状态 */}
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="react">React</option>
            <option value="redux">Redux</option>
            <option value="mobx">MobX</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

3. 复选框和单选框

复选框是类型为 checkbox 的 input 元素，单选框是类型为 radio 的 input 元素，它们的受控方式不同于类型为 text 的 input 元素。通常，复选框和单选框的值是不变的，需要改变的是它们的 checked 状态，因此 React 控制的属性不再是 value 属性，而是 checked 属性。例如：

```
class ReactStackForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = { react: false, redux: false, mobx: false };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  // 监听复选框变化，设置复选框的 checked 状态
  handleChange(event) {
    this.setState({ [event.target.name]: event.target.checked });
  }
```

```
  // 表单提交的响应函数
  handleSubmit(event) {
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        {/* 设置 3 个复选框 */}
        <label>React
        <input
          type="checkbox"
          name="react"
          value="react"
          checked={this.state.react}
          onChange={this.handleChange}
        />
        </label>
        <label>Redux
        <input
          type="checkbox"
          name="redux"
          value="redux"
          checked={this.state.redux}
          onChange={this.handleChange}
        />
        </label>
        <label>MobX
        <input
          type="checkbox"
          name="mobx"
          value="mobx"
          checked={this.state.mobx}
          onChange={this.handleChange}
        />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

上面的例子中，input 的 value 是不变的，onChange 事件改变的是 input 的 checked 属性。单选

框的用法和复选框相似，读者可自行尝试使用。

下面为 BBS 项目添加表单元素，让每一个帖子的标题支持编辑功能。本节项目源代码的目录为/chapter-02/bbs-components-form。修改后的 PostItem 如下：

```
class PostItem extends Component {
  constructor(props) {
    super(props);
    this.state = {
      editing: false,     // 帖子是否处于编辑态
      post: props.post
    };
    this.handleVote = this.handleVote.bind(this);
    this.handleEditPost = this.handleEditPost.bind(this);
    this.handleTitleChange = this.handleTitleChange.bind(this);
  }

  componentWillReceiveProps(nextProps) {
    // 父组件更新 post 后，更新 PostItem 的 state
    if (this.props.post !== nextProps.post) {
      this.setState({
        post: nextProps.post
      });
    }
  }
  // 处理点赞事件
  handleVote() {
    this.props.onVote(this.props.post.id);
  }
  // 保存/编辑按钮点击后的逻辑
  handleEditPost() {
    const editing = this.state.editing;
    // 当前处于编辑态，调用父组件传递的 onSave 方法保存帖子
    if (editing) {
      this.props.onSave({
        ...this.state.post,
        date: this.getFormatDate()
      });
    }
    this.setState({
      editing: !editing
    });
  }
  // 处理标题 textarea 值的变化
```

```
handleTitleChange(event) {
  const newPost = { ...this.state.post, title: event.target.value };
  this.setState({
    post: newPost
  });
}

getFormatDate() {
  //省略
}

render() {
  const { post } = this.state;
  return (
    <li className="item">
      <div className="title">
        {this.state.editing
          ? <form>
              <textarea
                value={post.title}
                onChange={this.handleTitleChange}
              />
            </form>
          : post.title}
      </div>
      <div>
        创建人：<span>{post.author}</span>
      </div>
      <div>
        创建时间：<span>{post.date}</span>
      </div>
      <div className="like">
        <span>
          <img alt="vote" src={like} onClick={this.handleVote} />
        </span>
        <span>
          {post.vote}
        </span>
      </div>
      <div>
        <button onClick={this.handleEditPost}>
          {this.state.editing ? "保存" : "编辑"}
        </button>
```

```
        </div>
      </li>
    );
  }
}
```

当点击编辑状态的 button 时，帖子的标题会使用 textarea 展示，此时标题处于可编辑状态，当再次点击 button 时，会执行保存操作，PostItem 通过 onSave 属性调用父组件 PostList 的 handleSave 方法，将更新后的 Post（标题和时间）保存到 PostList 的 state 中。PostList 中的修改如下：

```
class PostList extends Component {
  /** 省略其余代码 **/

  // 保存帖子
  handleSave(post) {
    // 根据 post 的 id，过滤出当前要更新的 post
    const posts = this.state.posts.map(item => {
      const newItem = item.id === post.id ? post : item;
      return newItem;
    })
    this.setState({
      posts
    })
  }

  render() {
    return (
      <div className='container'>
        <h2>帖子列表</h2>
        <ul>
          {this.state.posts.map(item =>
            <PostItem
              key = {item.id}
              post = {item}
              onVote = {this.handleVote}
              onSave = {this.handleSave}
            />
          )}
        </ul>
      </div>
    );
  }
}
```

2.6.2　非受控组件

使用受控组件虽然保证了表单元素的状态也由 React 统一管理，但需要为每个表单元素定义 onChange 事件的处理函数，然后把表单状态的更改同步到 React 组件的 state，这一过程是比较烦琐的，一种可替代的解决方案是使用非受控组件。非受控组件指表单元素的状态依然由表单元素自己管理，而不是交给 React 组件管理。使用非受控组件需要有一种方式可以获取到表单元素的值，React 中提供了一个特殊的属性 ref，用来引用 React 组件或 DOM 元素的实例，因此我们可以通过为表单元素定义 ref 属性获取元素的值。例如：

```
class SimpleForm extends Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleSubmit(event) {
    // 通过 this.input 获取到 input 元素的值
    alert('The title you submitted was ' + this.input.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          title:
          {/* this.input 指向当前 input 元素 */}
          <input type="text" ref={(input) => this.input = input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

ref 的值是一个函数，这个函数会接收当前元素作为参数，即例子中的 input 参数指向的是当前元素。在函数中，我们把 input 赋值给了 this.input，进而可以在组件的其他地方通过 this.input 获取这个元素。

在使用非受控组件时，我们常常需要为相应的表单元素设置默认值，但是无法通过表单元素的 value 属性设置，因为非受控组件中，React 无法控制表单元素的 value 属性，这也就意味着一旦在非受控组件中定义了 value 属性的值，就很难保证后续表单元素的值的正确性。这种情况下，我们可以使用 defaultValue 属性指定默认值：

```
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          title:
          <input defaultValue="something" type="text" ref={(input) =>
this.input = input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
```

上面的例子，defaultValue 设置的默认值为 something，而后续值的更改则由自己控制。类似地，select 元素和 textarea 元素也支持通过 defaultValue 设置默认值，<input type="checkbox"> 和<input type="radio">则支持通过 defaultChecked 属性设置默认值。

非受控组件看似简化了操作表单元素的过程，但这种方式破坏了 React 对组件状态管理的一致性，往往容易出现不容易排查的问题，因此非特殊情况下，不建议大家使用。

2.7 本章小结

本章详细介绍了 React 的主要特性及其用法。React 通过 JSX 语法声明界面 UI，将界面 UI 和它的逻辑封装到同一个 JS 文件中。组件是 React 的核心，根据组件的外部接口 props 和内部接口 state 完成自身 UI 的渲染。使用组件时，需要理解它的生命周期，借助不同的生命周期方法，组件可以实现复杂逻辑。组件在渲染列表数据时，要注意 key 的使用，在事件处理时，要注意事件名和事件处理函数的写法。最后介绍了 React 中表单元素的用法，表单元素的使用方式分为受控组件和非受控组件。

第 3 章

React 16 新特性

React 16 是 Facebook 在 2017 年 9 月发布的 React 最新版本。React 16 基于代号为“Fiber”的新架构实现，几乎对 React 的底层代码进行了重写，但对外的 API 基本不变，所以开发者可以几乎无缝地迁移到 React 16。此外，基于新的架构，React 16 实现了许多新特性。

3.1 render 新的返回类型

React 16 之前，render 方法必须返回单个元素。现在，render 方法支持两种新的返回类型：数组（由 React 元素组成）和字符串。定义一个 ListComponent 组件，它的 render 方法返回数组：

```
class ListComponent extends Component {
  render() {
    return [
      <li key="A">First item</li>,
      <li key="B">Second item</li>,
      <li key="C">Third item</li>
    ];
  }
}
```

再定义一个 StringComponent 组件，它的 render 方法返回字符串：

```
class StringComponent extends Component {
  render() {
```

```
    return "Just a strings";
  }
}
```

App 组件的 render 方法渲染 ListComponent 和 StringComponent：

```
export default class App extends Component {
  render() {
    return [
      <ul>
        <ListComponent />
      </ul>,
      <StringComponent />
    ];
  }
}
```

本节项目源代码的目录为/chapter-03/react16-render。

3.2 错误处理

React 16 之前，组件在运行期间如果执行出错，就会阻塞整个应用的渲染，这时候只能刷新页面才能恢复应用。React 16 引入了新的错误处理机制，默认情况下，当组件中抛出错误时，这个组件会从组件树中卸载，从而避免整个应用的崩溃。这种方式比起之前的处理方式有所进步，但用户体验依然不够友好。React 16 还提供了一种更加友好的错误处理方式——错误边界（Error Boundaries）。错误边界是能够捕获子组件的错误并对其做优雅处理的组件。优雅的处理可以是输出错误日志、显示出错提示等，显然这比直接卸载组件要更加友好。

定义了 componentDidCatch(error, info)这个方法的组件将成为一个错误边界，现在我们创建一个组件 ErrorBoundary：

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  componentDidCatch(error, info) {
    // 显示错误 UI
    this.setState({ hasError: true });
    // 同时输出错误日志
    console.log(error, info);
  }
```

```
  render() {
    if (this.state.hasError) {
      return <h1>Oops, something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

然后在 App 中使用 ErrorBoundary：

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      user: { name: "react" }
    };
  }
  // 将 user 置为 null，模拟异常
  onClick = () => {
    this.setState({ user: null });
  };

  render() {
    return (
      <div>
        <ErrorBoundary>
          <Profile user={this.state.user} />
        </ErrorBoundary>
        <button onClick={this.onClick}>更新</button>
      </div>
    );
  }
}

const Profile = ({ user }) => <div>name: {user.name}</div>;
```

点击更新按钮后，Profile 接收到的属性 user 为 null，程序会抛出 TypeError，这个错误会被 ErrorBoundary 捕获，并在界面上显示出错提示。注意，使用 create-react-app 创建的项目，当程序发生错误时，create-react-app 会在页面上创建一个浮层显示错误信息，要观察 ErrorBoundary 的正确效果，需要先关闭错误浮层。

本节项目源代码的目录为/chapter-03/react16-error-boundary。

3.3　Portals

React 16 的 Portals 特性让我们可以把组件渲染到当前组件树以外的 DOM 节点上，这个特性典型的应用场景是渲染应用的全局弹框，使用 Portals 后，任意组件都可以将弹框组件渲染到根节点上，以方便弹框的显示。Portals 的实现依赖 ReactDOM 的一个新的 API：

```
ReactDOM.createPortal(child, container)
```

第一个参数 child 是可以被渲染的 React 节点，例如 React 元素、由 React 元素组成的数组、字符串等，container 是一个 DOM 元素，child 将被挂载到这个 DOM 节点。

我们创建一个 Modal 组件，Modal 使用 ReactDOM.createPortal()在 DOM 根节点上创建一个弹框：

```
class Modal extends Component {
  constructor(props) {
    super(props);
   // 根节点下创建一个 div 节点
    this.container = document.createElement("div");
    document.body.appendChild(this.container);
  }

  componentWillUnmount() {
    document.body.removeChild(this.container);
  }

  render() {
   // 创建的 DOM 树挂载到 this.container 指向的 div 节点下面
    return ReactDOM.createPortal(
      <div className="modal">
        <span className="close" onClick={this.props.onClose}>
          &times;
        </span>
        <div className="content">
          {this.props.children}
        </div>
      </div>,
      this.container
    );
  }
}
```

在 App 中使用 Modal：

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = { showModal: true };
  }
  // 关闭弹框
  closeModal = () => {
    this.setState({ showModal: false });
  };

  render() {
    return (
      <div>
        <h2>Dashboard</h2>
        {this.state.showModal && (
          <Modal onClose={this.closeModal}>Modal Dialog</Modal>
        )}
      </div>
    );
  }
}

export default App;
```

本节项目源代码的目录为/chapter-03/react16-portals。

3.4　自定义 DOM 属性

React 16 之前会忽略不识别的 HTML 和 SVG 属性，现在 React 会把不识别的属性传递给 DOM 元素。例如，React 16 之前，下面的 React 元素

```
<div custom-attribute="something" />
```

在浏览器中渲染出的 DOM 节点为：

```
<div />
```

而 React 16 渲染出的 DOM 节点为：

```
<div custom-attribute="something" />
```

本节项目源代码的目录为/chapter-03/react16-custom-dom。

3.5 本章小结

本章介绍了 React 16 的新特性，主要包括 render 方法新支持的返回类型、新的错误处理机制和 Error Boundary 组件、可以将组件挂载到任意 DOM 树的 Portals 特性以及自定义 DOM 属性的支持。这些只是 React 16 常用的特性，除此之外，React 16 还有其他的新特性，例如 setState 传入 null 时不会再触发组件更新、更加高效的服务器端渲染方式等。相信基于新的 Fiber 架构，React 16 还会推出更多更强大的特性。

第 2 篇　进阶篇

用好React，你必须要知道的那些事

第 4 章

深入理解组件

在第 2 章中已经介绍了 React 组件的基本用法，本章将继续探究 React 组件，从组件的 state、组件与服务器通信、组件通信、组件的 ref 属性 4 个方面深入讲解组件，帮助读者在项目中正确地使用组件。

4.1 组件 state

4.1.1 设计合适的 state

组件 state 必须能代表一个组件 UI 呈现的完整状态集，即组件的任何 UI 改变都可以从 state 的变化中反映出来；同时，state 还必须代表一个组件 UI 呈现的最小状态集，即 state 中的所有状态都用于反映组件 UI 的变化，没有任何多余的状态，也不应该存在通过其他状态计算而来的中间状态。

我们通过一个例子来解释上面的定义。假设需要开发一个购物车组件，需要展示的信息有购买的物品列表以及物品的总金额。设计一个错误的 state：

```
// 错误的 state 示例
{
  purchaseList:[],
  totalCost: 0
}
```

这里的 state 是初始状态，因此 purchaseList 初始化为一个空数组，totalCost 初始化为 0，这个

state 的设计确实可以满足组件 UI 呈现的完整状态集这一条件，但是它包含一个无用的状态 totalCost，因为 totalCost 可以根据购买的每一项物品的价格和数量计算得出，所以有了 purchaseList，就可以计算出 totalCost，totalCost 属于中间状态，可以省略。

state 所代表的一个组件 UI 呈现的完整状态集又可以分成两类数据：用作渲染组件时使用到的数据的来源以及用作组件 UI 展现形式的判断依据。例如，下面的 Hello 组件定义了 user 和 display 作为组件 state，user 是组件最终要在界面上呈现的数据，而 display 决定了<h1>标签是否需要渲染，是组件 UI 呈现形式的判断依据。

```
class Hello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      user : 'React',
      display: true
    }
  }

  render() {
    return (
      <div>
        {
          this.state.display ?
          <h1>Hello, {this.state.user}</h1> : null
        }
      </div>
    );
  }
}
```

state 还容易和 props 以及组件的普通属性混淆。这是我们第一次提到组件的普通属性，所以先明确一下组件普通属性的定义。我们的组件都是使用 ES6 的 class 定义的，所以组件的属性其实也就是 class 的属性（更确切的说法是 class 实例化对象的属性，但因为 JavaScript 本质上是没有类的定义的，class 只不过是 ES6 提供的语法糖，所以这里模糊化类和对象的区别）。在 ES6 中，可以使用 this.{属性名}定义一个 class 的属性，也可以说属性是直接挂载到 this 下的变量。因此，state、props 实际上也是组件的属性，只不过它们是 React 为我们在 Component class 中预定义好的属性。除了 state、props 以外的其他组件属性称为组件的普通属性。

假设一个组件需要显示当前时间，并且这个时间每秒都会自动更新，这个组件内就需要定义一个计时器，在这个计时器中每隔 1 秒更新一次组件的 state。这个计时器变量并不适合定义到组件的 state 中，因为它并不代表组件 UI 呈现状态，它只是用来更改组件的 state，这时就到了组件的普通属性发挥作用的时候了。例如，下面的代码为 Hello 组件定义了 timer 属性，用来定时更新组件状态：

```
class Hello extends React.Component {
  constructor(props) {
    super(props);
    this.timer = null;   //普通属性
    this.state = {
      date : new Date()
    }
    this.updateDate = this.updateDate.bind(this);
  }

  componentDidMount() {
    this.timer = setInterval(this.updateDate, 1000)
  }

  componentWillUnmount() {
    clearInterval(this.timer);
  }

  updateDate(){
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello</h1>
        <h1>{this.state.date.toString()}</h1>
      </div>
    );
  }
}
```

因此，当我们在组件中需要用到一个变量，并且它与组件的渲染无关时，就应该把这个变量定义为组件的普通属性，直接挂载到 this 下，而不是作为组件的 state。还有一个更加直观的判断方法，就是看组件 render 方法中有没有使用到这个变量，如果没有，它就是一个普通属性。

state 和 props 又有什么区别呢？state 和 props 都直接和组件的 UI 渲染有关，它们的变化都会触发组件重新渲染，但 props 对于使用它的组件来说是只读的，是通过父组件传递过来的，要想修改 props，只能在父组件中修改；而 state 是组件内部自己维护的状态，是可变的。

总结一下，组件中用到的一个变量是不是应该作为 state 可以通过下面的 4 条依据进行判断：

（1）这个变量是否通过 props 从父组件中获取？如果是，那么它不是一个状态。

（2）这个变量是否在组件的整个生命周期中都保持不变？如果是，那么它不是一个状态。

（3）这个变量是否可以通过其他状态（state）或者属性（props）计算得到？如果是，那么它不是一个状态。

（4）这个变量是否在组件的 render 方法中使用？如果**不是**，那么它不是一个状态。这种情况下，这个变量更适合定义为组件的一个普通属性。

4.1.2　正确修改 state

state 可以通过 this.state.{属性}的方式直接获取，但当修改 state 时，往往有很多陷阱需要注意。下面介绍常见的三种陷阱：

1. 不能直接修改 state

直接修改 state，组件并不会重新触发 render。例如：

```
// 错误
this.state.title = 'React';
```

正确的修改方式是使用 setState()：

```
// 正确
this.setState({title: 'React'});
```

2. state 的更新是异步的

调用 setState 时，组件的 state 并不会立即改变，setState 只是把要修改的状态放入一个队列中，React 会优化真正的执行时机，并且出于性能原因，可能会将多次 setState 的状态修改合并成一次状态修改。所以不要依赖当前的 state，计算下一个 state。当真正执行状态修改时，依赖的 this.state 并不能保证是最新的 state，因为 React 会把多次 state 的修改合并成一次，这时 this.state 还是这几次 state 修改前的 state。另外，需要注意的是，同样不能依赖当前的 props 计算下一个状态，因为 props 的更新也是异步的。

举个例子，对于一个电商类应用，在购物车中，点击一次购买数量按钮，购买的数量就会加 1，如果连续点击两次按钮，就会连续调用两次 this.setState({quantity: this.state.quantity + 1})，在 React 合并多次修改为一次的情况下，相当于等价执行了如下代码：

```
Object.assign(
  previousState,
  {quantity: this.state.quantity + 1},
  {quantity: this.state.quantity + 1}
)
```

于是，后面的操作覆盖前面的操作，最终购买的数量只增加 1。

如果有这样的需求，可以使用另一个接收一个函数作为参数的 setState，这个函数有两个参数，第一个是当前最新状态（本次组件状态修改生效后的状态）的前一个状态 preState（本次组件状态修改前的状态），第二个参数是当前最新的属性 props。代码如下：

```
// 正确
this.setState((preState, props) => ({
  counter: preState.quantity + 1;
}))
```

3. state 的更新是一个合并的过程

当调用 setState 修改组件状态时，只需要传入发生改变的 state，而不是组件完整的 state，因为组件 state 的更新是一个合并的过程。例如，一个组件的状态为：

```
this.state = {
  title : 'React',
  content : 'React is an wonderful JS library!'
}
```

当只需要修改状态 title 时，将修改后的 title 传给 setState 即可：

```
this.setState({title: 'Reactjs'});
```

React 会合并新的 title 到原来的组件状态中，同时保留原有的状态 content，合并后的 state 为：

```
{
  title : 'Reactjs',
  content : 'React is an wonderful JS library!'
}
```

4.1.3 state 与不可变对象

React 官方建议把 state 当作不可变对象，一方面，直接修改 this.state，组件并不会重新 render；另一方面，state 中包含的所有状态都应该是不可变对象。当 state 中的某个状态发生变化时，应该重新创建这个状态对象，而不是直接修改原来的状态。那么，当状态发生变化时，如何创建新的状态呢？根据状态的类型可以分成以下三种情况：

1. 状态的类型是不可变类型（数字、字符串、布尔值、null、undefined）

这种情况最简单，因为状态是不可变类型，所以直接给要修改的状态赋一个新值即可。例如要修改 count（数字类型）、title（字符串类型）、success（布尔类型）三个状态：

```
this.setState({
  count: 1,
  title: 'React',
  success: true
})
```

2. 状态的类型是数组

例如有一个数组类型的状态 books，当向 books 中增加一本书时，可使用数组的 concat 方法或 ES6 的数组扩展语法（spread syntax）：

```
// 方法一：使用 preState、concat 创建新数组
this.setState(preState => ({
  books: preState.books.concat(['React Guide']);
}))

// 方法二：ES6 spread syntax
this.setState(preState => ({
  books: [...preState.books, 'React Guide'];
}))
```

当从 books 中截取部分元素作为新状态时，可使用数组的 slice 方法：

```
this.setState(preState => ({
  books: preState.books.slice(1,3);
}))
```

当从 books 中过滤部分元素后，作为新状态时，可使用数组的 filter 方法：

```
this.setState(preState => ({
  books: preState.books.filter(item => {
    return item !== 'React';
  });
}))
```

注意，不要使用 push、pop、shift、unshift、splice 等方法修改数组类型的状态，因为这些方法都是在原数组的基础上修改的，而 concat、slice、filter 会返回一个新的数组。

3. 状态的类型是普通对象（不包含字符串、数组）

（1）使用 ES6 的 Object.assgin 方法：

```
this.setState(preState => ({
  owner: Object.assign({}, preState.owner, {name: 'Jason'});
}))
```

（2）使用对象扩展语法（object spread properties）：

```
this.setState(preState => ({
  owner: {...preState.owner, name: 'Jason'};
}))
```

总结一下，创建新的状态对象的关键是，避免使用会直接修改原对象的方法，而是使用可以返回一个新对象的方法。当然，也可以使用一些 Immutable 的 JS 库（如 Immutable.js）实现类似的效果。

为什么 React 推荐组件的状态是不可变对象呢？一方面是因为对不可变对象的修改会返回一个新对象，不需要担心原有对象在不小心的情况下被修改导致的错误，方便程序的管理和调试；另一方面是出于性能考虑，当对象组件状态都是不可变对象时，在组件的 shouldComponentUpdate 方法中仅需要比较前后两次状态对象的引用就可以判断状态是否真的改变，从而避免不必要的 render 调用。在第 5 章会详细介绍这部分内容。

4.2 组件与服务器通信

React 关注的是 UI 的分离、视图的组件化，对于组件如何与服务器端 API 通信，React 官方并没有给出太多指导。但是，几乎所有应用都避免不了和服务器端 API 通信。这就给很多 React 的使用者带来了困惑，React 中的组件到底应该如何优雅地和服务器通信呢？本节将结合实践对这个问题的解决方法给出建议。

首先需要明确一点，本节讨论的组件与服务器通信特指组件从服务器上获取数据，不包含组件向服务器提交数据的情况。组件向服务器提交数据一定是由组件 UI 的某一事件触发的，比如提交了一个表单、点击了一个元素等，所以只要在监听相应事件的回调函数中执行向服务器提交数据的逻辑即可，一般不会有疑问。但组件从服务器上获取数据，情况就要复杂得多。

4.2.1 组件挂载阶段通信

React 组件的正常运转本质上是组件不同生命周期方法的有序执行，因此组件与服务器的通信也必定依赖组件的生命周期方法。我们先来看一下组件在挂载阶段如何与服务器通信。

定义一个 UserListContainer 组件，需要从服务器获取用户列表：

```
class UserListContainer extends React.Component{

 /** 省略无关代码 **/

  componentDidMount() {
    var that = this;
    fetch('/path/to/user-api').then(function(response) {
      response.json().then(function(data) {
        that.setState({users: data})
      });
    });
  }
}
```

UserListContainer 是在 componentDidMount 中与服务器进行通信的，这时候组件已经挂载，真实 DOM 也已经渲染完成，是调用服务器 API 最安全的地方，也是 React 官方推荐的进行服务器通信的地方。

除了 componentDidMount 外，在 componentWillMount 中进行服务器通信也是比较常见的一种方式。代码如下：

```
class UserListContainer extends React.Component{

 /** 省略无关代码 **/
```

```
    componentWillMount() {
      var that = this;
      fetch('/path/to/user-api').then(function(response) {
        response.json().then(function(data) {
          that.setState({users: data})
        });
      });
    }
  }
```

componentWillMount 会在组件被挂载前调用，因此从时间上来讲，在 componentWillMount 中执行服务器通信要早于在 componentDidMount 中执行，执行得越早意味着服务器数据越能更快地返回组件。这也是很多人青睐在 componentWillMount 中执行服务器通信的重要原因。但实际上，componentWillMount 与 componentDidMount 执行的时间差微乎其微，完全可以忽略不计。

componentDidMount 是执行组件与服务器通信的最佳地方，原因主要有两个：

（1）在 componentDidMount 中执行服务器通信可以保证获取到数据时，组件已经处于挂载状态，这时即使要直接操作 DOM 也是安全的，而 componentWillMount 无法保证这一点。

（2）当组件在服务器端渲染时（本书不涉及服务器渲染内容），componentWillMount 会被调用两次，一次是在服务器端，另一次是在浏览器端，而 componentDidMount 能保证在任何情况下只会被调用一次，从而不会发送多余的数据请求。

有些开发人员会在组件的构造函数中执行服务器通信，一般情况下，这种方式也可以正常工作。但是，构造函数的意义是执行组件的初始化工作，如设置组件的初始状态，并不适合做数据请求这类有“副作用”的工作。因此，不推荐在构造函数中执行服务器通信。

4.2.2　组件更新阶段通信

组件在更新阶段常常需要再次与服务器通信，获取服务器上的最新数据。例如，组件需要以 props 中的某个属性作为与服务器通信时的请求参数，当这个属性值发生更新时，组件自然需要重新与服务器通信。回想 2.3 节中对组件生命周期的介绍，不难发现 componentWillReceiveProps 非常适合做这个工作。假设 UserListContainer 在获取用户列表时还需要一个参数 category，用来根据用户的职业做筛选，category 这个参数是从 props 中获取的，实现代码如下：

```
class UserListContainer extends React.Component{

  /** 省略无关代码 **/

  componentWillReceiveProps(nextProps) {
    if(nextProps.category !== this.props.category) {
      fetch('/path/to/user-api?category='+ nextProps.category).
then(function(response) {
        response.json().then(function(data) {
```

```
          that.setState({users: data})
        });
      });
    }
  }

}
```

这里还有一个地方要注意，在执行 fetch 请求时，要先对新老 props 中的 category 做比较，只有不一致才说明 category 有了更新，才需要重新进行服务器通信。componentWillReceiveProps 的执行并不能保证 props 一定发生了修改。

4.3 组件通信

一个 React 应用是由许多个组件像搭积木一样搭建而成的，只要应用不是完全由展示组件组成的，组件之间就难免需要进行通信。其实，前面一些内容已经涉及组件的通信，只是我们并没有刻意强调这个概念。下面系统地介绍组件是如何进行通信的。

4.3.1 父子组件通信

父子组件通信是最常见的通信形式，例如 4.2 节的 UserListContainer 组件获取到的用户数据需要通过 UserList 组件展示，这时 UserListContainer 和 UserList 就存在父子组件通信。UserListContainer 作为父组件，将获取到的用户信息通过子组件 UserList 的 props 传递给 UserList。所以父组件向子组件通信是通过父组件向子组件的 props 传递数据完成的。代码如下：

```
class UserList extends React.Component{

  render() {
    return (
      <div>
        <ul className="user-list">
          {this.props.users.map(function(user) {
            return (
              <li key={user.id}>
                <span>{user.name}</span>
              </li>
            );
          })}
        </ul>
      </div>
    )
  }
```

```
}

import UserList from './UserList'

class UserListContainer extends React.Component{

  constructor(props){
    super(props);
    this.state = {
      users: []
    }
  }

  componentDidMount() {
    var that = this;
    fetch('/path/to/user-api').then(function(response) {
      response.json().then(function(data) {
        that.setState({users: data})
      });
    });
  }

  render() {
    return (
      {/* 通过 props 传递 users */}
      <UserList users={this.state.users} />
    )
  }
}
```

当子组件需要向父组件通信时，又该怎么做呢？答案依然是 props。父组件可以通过子组件的 props 传递给子组件一个回调函数，子组件在需要改变父组件数据时，调用这个回调函数即可。下面为 UserList 再增加一个添加新用户的功能：

```
class UserList extends React.Component{

  constructor(props){
    super(props);
    this.state = {
      newUser : ''
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleClick = this.handleClick.bind(this);
  }
```

```
  handleChange(e) {
    this.setState({newUser: e.target.value});
  }
  // 通过 props 调用父组件的方法新增用户
  handleClick() {
    if(this.state.newUser && this.state.newUser.length > 0) {
      this.props.onAddUser(this.state.newUser);
    }
  }

  render() {
    return (
      <div>
        <ul className="user-list">
          {this.props.users.map(function(user) {
            return (
              <li key={user.id}>
                <span>{user.name}</span>
              </li>
            );
          })}
        </ul>
        <input onChange={this.handleChange} value={this.state.newUser} />
        <button onClick={this.handleClick}>新增</button>
      </div>
    )
  }
}
```

在 input 内输入新用户的名称，然后点击新增按钮，调用 handleClick 方法，在 handleClick 内部，调用通过 props 传递过来的 onAddUser 执行保存用户的逻辑。下面来看一下 onAddUser 在 UserListContainer 中的实现：

```
import UserList from './UserList'

class UserListContainer extends React.Component{

  constructor(props){
    super(props);
    this.state = {
      users: []
    }
    this.handleAddUser = this.handleAddUser.bind(this);
  }

  componentDidMount() {
```

```
    var that = this;
    fetch('/path/to/user-api').then(function(response) {
      response.json().then(function(data) {
        that.setState({users: data})
      });
    });
  }
  // 新增用户
  handleAddUser(user) {
    var that = this;
    fetch('/path/to/save-user-api',{
      method: 'POST',
      body: JSON.stringify({'username':user})
    }).then(function(response) {
      response.json().then(function(newUser) {
        // 将服务器端返回的新用户添加到 state 中
        that.setState((preState) => ({users: preState.users.concat
([newUser])}))
      });
    });
  }

  render() {
    return (
      {/* 通过 props 传递 users 和 handleAddUser 方法 */}
      <UserList users={this.state.users}
         onAddUser={this.handleAddUser}
      />
    )
  }
}
```

子组件 UserList 通过调用 props.onAddUser 方法成功地将待新增的用户传递给父组件 UserListContainer 的 handleAddUser 方法执行保存操作，保存成功后，UserListContainer 会更新状态 users，从而又将最新的用户列表传递给 UserList。这一过程既包含子组件到父组件的通信，又包含父组件到子组件的通信，而通信的桥梁就是通过 props 传递的数据和回调方法。

4.3.2　兄弟组件通信

当两个组件不是父子关系但有相同的父组件时，称为兄弟组件。注意，这里的兄弟组件在整个组件树上并不一定处于同一层级，如图 4-1 所示的两种情况，B 和 C 都是兄弟组件，因为他们都有相同的父组件 A。

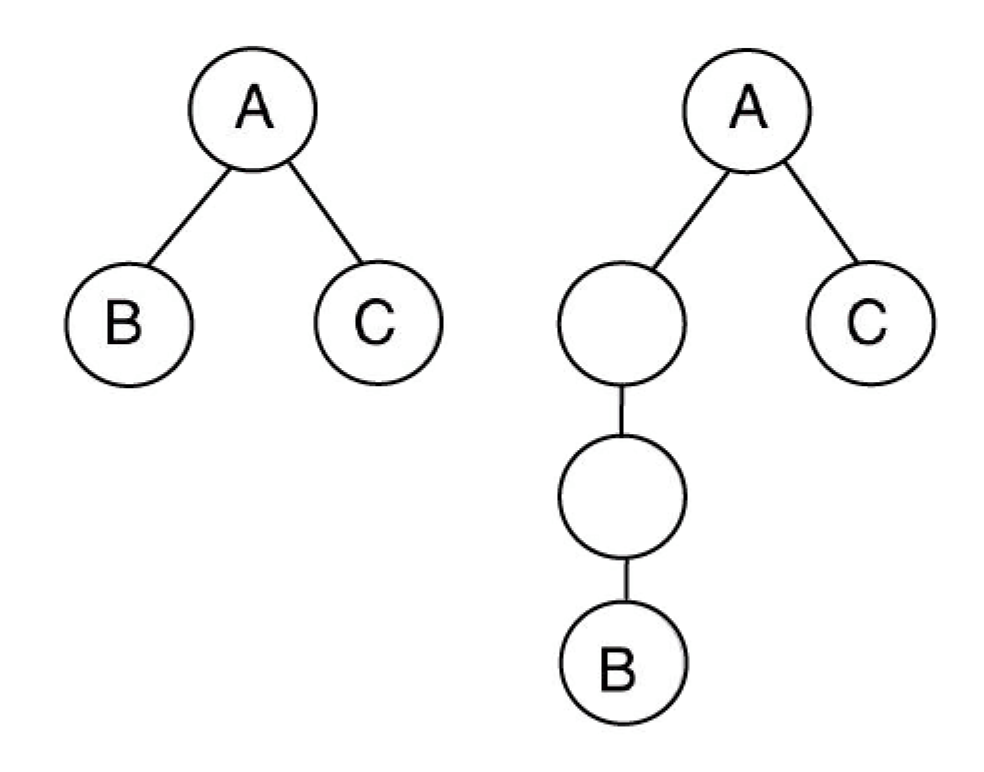


图 4-1

兄弟组件不能直接相互传送数据，需要通过状态提升的方式实现兄弟组件的通信，即把组件之间需要共享的状态保存到距离它们最近的共同父组件内，任意一个兄弟组件都可以通过父组件传递的回调函数来修改共享状态，父组件中共享状态的变化也会通过 props 向下传递给所有兄弟组件，从而完成兄弟组件之间的通信。

我们在 UserListContainer 中新增一个子组件 UserDetail，用于显示当前选中用户的详细信息，比如用户的年龄、联系方式、家庭地址等。这时，UserList 和 UserDetail 就成了兄弟组件，UserListContainer 是它们的共同父组件。当用户在 UserList 中点击一条用户信息时，UserDetail 需要同步显示该用户的详细信息，因此，可以把当前选中的用户 currentUser 保存到 UserListContainer 的状态中。

先来修改 UserList 组件：

```
class UserList extends React.Component{

  constructor(props){
    super(props);
    this.state = {
      newUser : ''
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleClick = this.handleClick.bind(this);
  }

  handleChange(e) {
    this.setState({newUser: e.target.value});
  }
  // 通过 props 调用父组件的方法新增用户
  handleClick() {
    if(this.state.newUser && this.state.newUser.length > 0) {
      this.props.onAddUser(this.state.newUser);
    }
  }
  // 通过 props 调用父组件的方法，设置当前选中的用户
```

```
  handleUserClick(userId) {
    this.props.onSetCurrentUser(userId);
  }

  render() {
    return (
      <div>
        <ul className="user-list">
          {this.props.users.map((user) => {
            return (
              <li key={user.id}
           {/*使用不同样式显示当前用户 */}
                className={(this.props.currentUserId === user.id) ? 'current' : ''}
                onClick={this.handleUserClick.bind(this, user.id)}>
                <span>{user.name}</span>
              </li>
            );
          })}
        </ul>
        <input onChange={this.handleChange} value={this.state.newUser} />
        <button onClick={this.handleClick}>新增</button>
      </div>
    )
  }
}
```

我们为 UserList 添加了处理点击用户项的回调函数 handleUserClick，还为当前处于选中状态的用户项添加了名为 current 的样式。

再来创建 UserDetail 组件：

```
function UserDetail(props) {
  return (
      <div>
        {props.currentUser ?
          (<div>用户姓名：{props.currentUser.name}</div>
          <div>用户年龄：{props.currentUser.age}</div>
          <div>用户联系方式：{props.currentUser.phone}</div>
          <div>家庭地址：{props.currentUser.address}</div>)
          : ''
        }
      </div>
      )
}
```

UserDetail 不需要维护自己的状态，因此最适合用函数组件来实现。

最后修改 UserListContainer 组件：

```
import UserList from './UserList'
import UserDetail from './UserDetail'

class UserListContainer extends React.Component{

  constructor(props){
    super(props);
    this.state = {
      users: [],
      currentUserId: null
    }
    this.handleAddUser = this.handleAddUser.bind(this);
    this.handleSetCurrentUser = this.handleSetCurrentUser.bind(this);
  }

  componentDidMount() {
    var that = this;
    fetch('/path/to/user-api').then(function(response) {
      response.json().then(function(data) {
        that.setState({users: data})
      });
    });
  }
  // 新增用户
  handleAddUser(user) {
    var that = this;
    fetch('/path/to/save-user-api',{
      method: 'POST',
      body: JSON.stringify({'username':user})
    }).then(function(response) {
      response.json().then(function(newUser) {
        that.setState((preState) => ({users: preState.users.concat
([newUser])}))
      });
    });
  }
  // 设置当前选中的用户
  handleSetCurrentUser(userId) {
    this.setState({
      currentUserId : userId
    });
  }
```

```
    render() {
    // 根据 currentUserId，筛选出当前用户对象
      const filterUsers = this.state.users.filter((user) => {user.id === 
this.state.currentUserId});
      const currentUser = filterUsers.length > 0 ? filterUsers[0] : null;
      return (
        <UserList users={this.state.users}
          currentUserId = {this.state.currentUserId}
          onAddUser = {this.handleAddUser}
          onSetCurrentUser = {this.handleSetCurrentUser}
        />
        <UserDetail currentUser = {currentUser} />
      )
    }
  }
```

UserListContainer 新增状态 currentUserId 用来标识当前选中的用户，这个状态正是 UserList 和 UserDetail 两个组件都要用到的状态，通过状态提升保存到它们共同的父组件 UserListContainer 中。同时，UserListContainer 通过 UserList 的 props 将修改 currentUserId 的回调函数传递给 UserList，使 UserList 可以在自身内部修改 currentUserId。

4.3.3　Context

当组件所处层级太深时，往往需要经过很多层的 props 传递才能将所需的数据或者回调函数传递给使用组件。这时，以 props 作为桥梁的组件通信方式便会显得很烦琐。例如，我们把 UserList 中新增用户的工作单独拆分到一个新的组件 UserAdd 中：

```
class UserAdd extends React.Component{

  constructor(props){
    super(props);
    this.state = {
      newUser : ''
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleClick = this.handleClick.bind(this);
  }

  handleChange(e) {
    this.setState({newUser: e.target.value});
  }
  // 通过 props 调用父组件的方法新增用户
  handleClick() {
    if(this.state.newUser && this.state.newUser.length > 0) {
      this.props.onAddUser(this.state.newUser);
```

```
    }
  }

  render() {
    return (
      <div>
        <input onChange={this.handleChange} value={this.state.newUser} />
        <button onClick={this.handleClick}>新增</button>
      </div>
    )
  }
}
```

同时，修改原有的 UserList 组件：

```
import UserAdd from './UserAdd'

class UserList extends React.Component{
  // 通过 props 调用父组件的方法，设置当前用户
  handleUserClick(userId) {
    this.props.onSetCurrentUser(userId);
  }

  render() {
    return (
      <div>
        <ul className="user-list">
          {this.props.users.map((user) => {
            return (
              <li key={user.id}
                className={(this.props.currentUserId === user.id) ? 'current' : ''}
                onClick={this.handleUserClick.bind(this, user.id)}>
                <span>{user.name}</span>
              </li>
            );
          })}
        </ul>
        {/* 传递 UserListContainer 的 handleAddUser 方法 */}
        <UserAdd onAddUser = {this.props.onAddUser} />
      </div>
    )
  }
}
```

可以发现，UserListContainer 中处理添加用户的函数 handleAddUser 经过 UserList 和 UserAdd 两个层级的 props 传递才到达 UserAdd 组件中。当应用更加复杂时，组件的层级会更多，组件通信

就需要经过更多层级的传递，组件通信会变得非常麻烦。幸好，React 提供了一个 context 上下文，让任意层级的子组件都可以获取父组件中的状态和方法。创建 context 的方式是：在提供 context 的组件内新增一个 getChildContext 方法，返回 context 对象，然后在组件的 childContextTypes 属性上定义 context 对象的属性的类型信息。UserListContainer 是提供 context 的组件，改写如下：

```
class UserListContainer extends React.Component{

  /** 省略其余代码 **/

  // 创建 context 对象，包含 onAddUser 方法
  getChildContext() {
    return {onAddUser: this.handleAddUser};
  }
  // 新增用户
  handleAddUser(user) {
    this.setState((preState) => ({users: preState.users.concat([{'id':'c',
'name': 'cc' }])}))
  }

  render() {
    const filterUsers = this.state.users.filter((user) => {user.id =
this.state.currentUserId});
    const currentUser = filterUsers.length > 0 ? filterUsers[0] : null;
    return (
      <UserList users={this.state.users}
        currentUserId = {this.state.currentUserId}
        onSetCurrentUser = {this.handleSetCurrentUser}
      />
      <UserDetail currentUser = {currentUser} />
    )
  }
}

// 声明 context 的属性的类型信息
UserListContainer.childContextTypes = {
  onAddUser: PropTypes.func
};
```

UserListContainer 通过增加 getChildContext 和 childContextTypes 将 onAddUser 在组件树中自动向下传递，当任意层级的子组件需要使用时，只需要在该组件的 contextTypes 中声明使用的 context 属性即可。例如，UserAdd 需要使用 context 中的 onAddUser，代码如下：

```
class UserAdd extends React.Component{
```

```
    /**省略其余代码**/

    handleChange(e) {
      this.setState({newUser: e.target.value});
    }

    handleClick() {
      if(this.state.newUser && this.state.newUser.length > 0) {
        this.context.onAddUser(this.state.newUser);
      }
    }

    render() {
      return (
        <div>
          <input onChange={this.handleChange} value={this.state.newUser} />
          <button onClick={this.handleClick}>Add</button>
        </div>
      )
    }
  }

  // 声明要使用的 context 对象的属性
  UserAdd.contextTypes = {
    onAddUser: PropTypes.func
  };
```

增加 contextTypes 后，在 UserAdd 内部就可以通过 this.context.onAddUser 的方式访问 context 中的 onAddUser 方法。注意，这里的示例传递的是组件的方法，组件中的任意数据也可以通过 context 自动向下传递。另外，当 context 中包含数据时，如果要修改 context 中的数据，一定不能直接修改，而是要通过 setState 修改，组件 state 的变化会创建一个新的 context，然后重新传递给子组件。

虽然 context 给组件通信带来了便利，但过多使用 context 会让应用中的数据流变得混乱，而且 context 是一个实验性的 API，在未来的 React 版本中是可能被修改或者废弃的。所以，使用 context 一定要慎重。

4.3.4 延伸

前面介绍的三种组件通信方式都是依赖 React 组件自身的语法特性。其实，还有更多的方式可以来实现组件通信。我们可以使用消息队列来实现组件通信：改变数据的组件发起一个消息，使用数据的组件监听这个消息，并在响应函数中触发 setState 来改变组件状态。本质上，这是观察者模式的实现，我们可以通过引入 EventEmitter 或 Postal.js 等消息队列库完成这一过程。当应用更加复杂时，还可以引入专门的状态管理库实现组件通信和组件状态的管理，例如 Redux 和 MobX 是当前非常受欢迎的两种状态管理库。后面的章节中会对这两种状态解决方案做详细介绍。

4.4　特殊的 ref

在 2.6.2 节非受控组件中，已经使用过 ref 来获取表单元素。ref 不仅可以用来获取表单元素，还可以用来获取其他任意 DOM 元素，甚至可以用来获取 React 组件实例。在一些场景下，ref 的使用可以带来便利，例如控制元素的焦点、文本的选择或者和第三方操作 DOM 的库集成。但绝大多数场景下，应该避免使用 ref，因为它破坏了 React 中以 props 为数据传递介质的典型数据流。本节将介绍 ref 常用的使用场景。

4.4.1　在 DOM 元素上使用 ref

在 DOM 元素上使用 ref 是最常见的使用场景。ref 接收一个回调函数作为值，在组件被挂载或卸载时，回调函数会被调用，在组件被挂载时，回调函数会接收当前 DOM 元素作为参数；在组件被卸载时，回调函数会接收 null 作为参数。例如：

```
class AutoFocusTextInput extends React.Component {
  componentDidMount() {
    // 通过 ref 让 input 自动获取焦点
    this.textInput.focus();
  }

  render() {
    return (
      <div>
        <input
          type="text"
          ref={(input) => { this.textInput = input; }} />
      </div>
    );
  }
}
```

AutoFocusTextInput 中为 input 元素定义 ref，在组件挂载后，通过 ref 获取该 input 元素，让 input 自动获取焦点。如果不使用 ref，就难以实现这个功能。

4.4.2　在组件上使用 ref

React 组件也可以定义 ref，此时 ref 的回调函数接收的参数是当前组件的实例，这提供了一种在组件外部操作组件的方式。例如，在使用 AutoFocusTextInput 组件的外部组件 Container 中控制 AutoFocusTextInput：

```
class AutoFocusTextInput extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.blur = this.blur.bind(this);
  }

  componentDidMount() {
    // 通过 ref 让 input 自动获取焦点
    this.textInput.focus();
  }
  // 让 input 失去焦点
  blur() {
    this.textInput.blur();
  }

  render() {
    return (
      <div>
        <input
          type="text"
          ref={(input) => { this.textInput = input; }} />
      </div>
    );
  }
}

class Container extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // 通过 ref 调用 AutoFocusTextInput 组件的方法
    this.inputInstance.blur();
  }

  render() {
    return (
      <div>
       <AutoFocusTextInput ref={(input) => {this.inputInstance = input}}/>
       <button onClick={this.handleClick}>失去焦点</button>
      </div>
    );
  }
}
```

在 Container 组件中，我们通过 ref 获取到了 AutoFocusTextInput 组件的实例对象，并把它赋值给 Container 的 inputInstance 属性，这样就可以通过 inputInstance 调用 AutoFocusTextInput 中的 blur 方法，让已经处于获取焦点状态的 input 元素失去焦点。

注意，只能为类组件定义 ref 属性，而不能为函数组件定义 ref 属性，例如下面的写法是不起作用的：

```
function MyFunctionalComponent() {
  return <input />;
}

class Parent extends React.Component {
  render() {
    // ref 不生效
    return (
      <MyFunctionalComponent
        ref={(input) => { this.textInput = input; }} />
    );
  }
}
```

函数组件虽然不能定义 ref 属性，但这并不影响在函数组件内部使用 ref 来引用其他 DOM 元素或组件，例如下面的例子是可以正常工作的：

```
function MyFunctionalComponent() {
  let textInput = null;

  function handleClick() {
    textInput.focus();
  }

  return  (
    <div>
      <input
        type="text"
        ref={(input) => { textInput = input; }} />
      <button onClick={handleClick}>获取焦点</button>
    </div>
  )
}
```

4.4.3　父组件访问子组件的 DOM 节点

在一些场景下，我们可能需要在父组件中获取子组件的某个 DOM 元素，例如父组件需要知道这个 DOM 元素的尺寸或位置信息，这时候直接使用 ref 是无法实现的，因为 ref 只能获取子组件

的实例对象，而不能获取子组件中的某个 DOM 元素。不过，我们可以采用一种间接的方式获取子组件的 DOM 元素：在子组件的 DOM 元素上定义 ref，ref 的值是父组件传递给子组件的一个回调函数，回调函数可以通过一个自定义的属性传递，例如 inputRef，这样父组件的回调函数中就能获取到这个 DOM 元素。下面的例子中，父组件 Parent 的 inputElement 指向的就是子组件的 input 元素。

```
function Children(props) {
  // 子组件使用父组件传递的 inputRef，为 input 的 ref 赋值
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}

class Parent extends React.Component {
  render() {
    // 自定义一个属性 inputRef，值是一个函数
    return (
      <Children
        inputRef={el => this.inputElement = el}
      />
    );
  }
}
```

从这个例子中还可以发现，即使子组件是函数组件，这种方式同样有效。

4.5 本章小结

本章再次讨论 React 组件。首先，我们详细介绍了组件 state，包括 state 的设计、state 的修改以及 state 和不可变对象之间的关系；接着，我们介绍了组件与服务器通信，这是初学者常常产生困惑的地方，关键是要清楚应该在组件的哪些生命周期方法中进行服务器请求，组件之间的通信桥梁是 props，要注意父子组件通信时状态提升的情况，context 虽然能简化组件的通信，但它破坏了 React 组件的数据流，使用时要慎重；最后，我们介绍了 ref 的 3 种常见的使用场景，ref 也需要避免过度使用。

第 5 章

虚拟 DOM 和性能优化

React 之所以执行效率高，一个很重要的原因是它的虚拟 DOM 机制。React 应用常用的性能优化方法也大都与虚拟 DOM 机制相关。本章将先介绍虚拟 DOM 的概念和用于虚拟 DOM 结构比较的 Diff 算法，然后介绍 React 常用的性能优化方法和性能检测工具。

5.1 虚拟 DOM

虚拟 DOM 是和真实 DOM 相对应的，真实 DOM 也就是平时我们所说的 DOM，它是对结构化文本的抽象表达。在 Web 环境中，其实就是对 HTML 文本的一种抽象描述，每一个 HTML 元素对应一个 DOM 节点，HTML 元素的层级关系也会体现在 DOM 节点的层级上，所有的这些 DOM 节点构成一棵 DOM 树。

在传统的前端开发中，通过调用浏览器提供的一组 API 直接对 DOM 执行增删改查的操作。例如，通过 getElementById 查询一个 DOM 节点，通过 insertBefore 在某个节点前插入一个新的节点等。这些操作看似只执行了一条 JavaScript 语法，但它们的执行效率要比执行一条普通 JavaScript 语句慢得多，尤其是对 DOM 进行增删改操作，每一次对 DOM 的修改都会引起浏览器对网页的重新布局和重新渲染，而这个过程是很耗时的。这也是为什么前端性能优化中有一条原则：尽量减少 DOM 操作。

既然操作 DOM 效率低下，那么有什么办法可以解决这个问题呢？在软件开发中，有这么一句话：软件开发中遇到的所有问题都可以通过增加一层抽象而得以解决。DOM 效率低下的这个问题同样可以通过增加一层抽象解决。虚拟 DOM 就是这层抽象，建立在真实 DOM 之上，对真实 DOM

的抽象。这里需要注意，虚拟 DOM 并非 React 所独有的，它是一个独立的技术，只不过 React 使用了这项技术来提高自身性能。

虚拟 DOM 使用普通的 JavaScript 对象来描述 DOM 元素，对象的结构和 2.1 节中 React.createElement 方法使用的参数的结构类似，实际上，React 元素本身就是一个虚拟 DOM 节点。例如，下面是一个 DOM 结构：

```
<div className="foo">
    <h1>Hello React</h1>
</div>
```

可以用这样的一个 JavaScript 对象来表述：

```
{
  type: 'div',
  props: {
    className: 'foo',
    children: {
      type: 'h1',
      props:{
        children: 'Hello React'
      }
    }
  }
}
```

有了虚拟 DOM 这一层，当我们需要操作 DOM 时，就可以操作虚拟 DOM，而不操作真实 DOM，虚拟 DOM 是普通的 JavaScript 对象，访问 JavaScript 对象当然比访问真实 DOM 要快得多。到这里，大家可以发现，虚拟 DOM 并不是什么神奇的东西，它只是用来描述真实 DOM 的 JavaScript 对象而已。

5.2 Diff 算法

React 采用声明式的 API 描述 UI 结构，每次组件的状态或属性更新，组件的 render 方法都会返回一个新的虚拟 DOM 对象，用来表述新的 UI 结构。如果每次 render 都直接使用新的虚拟 DOM 来生成真实 DOM 结构，那么会带来大量对真实 DOM 的操作，影响程序执行效率。事实上，React 会通过比较两次虚拟 DOM 结构的变化找出差异部分，更新到真实 DOM 上，从而减少最终要在真实 DOM 上执行的操作，提高程序执行效率。这一过程就是 React 的调和过程（Reconciliation），其中的关键是比较两个树形结构的 Diff 算法。


注 意

在 Diff 算法中，比较的两方是新的虚拟 DOM 和旧的虚拟 DOM，而不是虚拟 DOM 和真实 DOM，只不过 Diff 的结果会更新到真实 DOM 上。

正常情况下，比较两个树形结构差异的算法的时间复杂度是 O(N^3)，这个效率显然是无法接受的。React 通过总结 DOM 的实际使用场景提出了两个在绝大多数实践场景下都成立的假设，基于这两个假设，React 实现了在 O(N)时间复杂度内完成两棵虚拟 DOM 树的比较。这两个假设是：

（1）如果两个元素的类型不同，那么它们将生成两棵不同的树。

（2）为列表中的元素设置 key 属性，用 key 标识对应的元素在多次 render 过程中是否发生变化。

下面介绍在不同情况下，React 具体是如何比较两棵树的差异的。React 比较两棵树是从树的根节点开始比较的，根节点的类型不同，React 执行的操作也不同。

1. 当根节点是不同类型时

从 div 变成 p、从 ComponentA 变成 ComponentB，或者从 ComponentA 变成 div，这些都是节点类型发生变化的情况。根节点类型的变化是一个很大的变化，React 会认为新的树和旧的树完全不同，不会再继续比较其他属性和子节点，而是把整棵树拆掉重建（包括虚拟 DOM 树和真实 DOM 树）。这里需要注意，虚拟 DOM 的节点类型分为两类：一类是 DOM 元素类型，比如 div、p 等；一类是 React 组件类型，比如自定义的 React 组件。在旧的虚拟 DOM 树被拆除的过程中，旧的 DOM 元素类型的节点会被销毁，旧的 React 组件实例的 componentWillUnmount 会被调用；在重建的过程中，新的 DOM 元素会被插入 DOM 树中，新的组件实例的 componentWillMount 和 componentDidMount 方法会被调用。重建后的新的虚拟 DOM 树又会被整体更新到真实 DOM 树中。这种情况下，需要大量 DOM 操作，更新效率最低。

2. 当根节点是相同的 DOM 元素类型时

如果两个根节点是相同类型的 DOM 元素，React 会保留根节点，而比较根节点的属性，然后只更新那些变化了的属性。例如：

```
<div className="foo" title="React" />

<div className="bar" title="React" />
```

React 比较这两个元素，发现只有 className 属性发生了变化，然后只更新虚拟 DOM 树和真实 DOM 树中对应节点的这一属性。

3. 当根节点是相同的组件类型时

如果两个根节点是相同类型的组件，对应的组件实例不会被销毁，只是会执行更新操作，同步变化的属性到虚拟 DOM 树上，这一过程组件实例的 componentWillReceiveProps()和 componentWillUpdate()会被调用。注意，对于组件类型的节点，React 是无法直接知道如何更新真实 DOM 树的，需要在组件更新并且 render 方法执行完成后，根据 render 返回的虚拟 DOM 结构决定如何更新真实 DOM 树。

比较完根节点后，React 会以同样的原则继续递归比较子节点，每一个子节点相对于其层级以下的节点来说又是一个根节点。如此递归比较，直到比较完两棵树上的所有节点，计算得到最终的差异，更新到 DOM 树中。

当一个节点有多个子节点时，默认情况下，React 只会按照顺序逐一比较两棵树上对应的子节点。例如，比较下面的两个节点，两棵树上的
first
和
second
子节点会分别被匹配，最终只会插入一个新的节点
third。

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

但如果在子节点的开始位置新增一个节点，情况就会变得截然不同。例如下面的例子，
third
插入子节点的第一个位置，React 会把第一棵树的
first
和第二棵树的
third
进行比较，把第一棵树的
second
和第二棵树的
first
进行比较，最后发现新增了一个
second
节点。这种比较方式会导致每一个节点都被修改。

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>third</li>
  <li>first</li>
  <li>second</li>
</ul>
```

为了解决这种低效的更新方式，React 提供了一个 key 属性。在 2.4 节已经介绍过，当渲染列表元素时，需要为每一个元素定义一个 key。这个 key 就是为了帮助 React 提高 Diff 算法的效率。当一组子节点定义了 key，React 会根据 key 来匹配子节点，在每次渲染之后，只要子节点的 key 值没有变化，React 就认为这是同一个节点。例如，为前面的例子定义 key：

```
<ul>
  <li key="first" >first</li>
  <li key="second">second</li>
</ul>

<ul>
  <li key="third">third</li>
  <li key="first">first</li>
  <li key="second">second</li>
</ul>
```

定义 key 之后，React 就能判断出<li key="third">third
这个节点是新增节点，<li key="first">first
和<li key="second">second
两个节点并没有发生改变，只是位置发生了变化而已。如此一来，React 只需要执行一次插入新节点的操作。这里同时揭露了另一个问题，尽量不要使用元素在列表中的索引值作为 key，因为列表中的元素顺序一旦发生改变，就可能导致大量的 key 失效，进而引起大量的修改操作。例如，下面的写法应该尽量避免：

```
<ul>
  {list.map((item, index) => <li key={index}>{item}</li> )}
</ul>
```

最后，需要提醒读者，本章介绍的 Diff 算法是 React 当前采用的实现方式，React 会不断改进 Diff 算法，以提高 DOM 比较和更新的效率。

5.3　性能优化

React 通过虚拟 DOM、高效的 Diff 算法等技术极大地提高了操作 DOM 的效率。在大多数场景下，我们是不需要考虑 React 程序的性能问题的，但只要是程序，总会有一些优化的措施。本章就来介绍一下 React 中常用的性能优化方式。

1. 使用生产环境版本的库

这是性能优化的一个基本原则，也是很容易被忽视的一个原则。我们使用 create-react-app 脚手架创建的项目，在以 npm run start 启动时，使用的 React 是开发环境版本的 React 库，包含大量警告消息，以帮助我们在开发过程中避免一些常见的错误，比如组件 props 类型的校验等。开发环境版本的库不仅体积更大，而且执行速度也更慢，显然不适合在生产环境中使用。那么，如何构建生产环境版本的 React 库呢？对于 create-react-app 脚手架创建的项目，只需要执行 npm run build，就会构建生产环境版本的 React 库。其原理是，一般第三方库都会根据 process.env.NODE_ENV 这个环境变量决定在开发环境和生产环境下执行的代码有哪些不同，当执行 npm run build 时，构建脚本会把 NODE_ENV 的值设置为 production，也就是会以生产环境模式编译代码。

如果不是使用 create-react-app 脚手架创建的项目，而是完全自己编写 Webpack 的构建配置，那么在执行生产环境的构建时，就需要在 Webpack 的配置项中包含以下插件的配置：

```
plugins: [
  new webpack.DefinePlugin({
    'process.env': {
      NODE_ENV: JSON.stringify('production')
    }
  }),
  new UglifyJSPlugin(),
  //...
]
```

当 NODE_ENV 等于 production 时，不仅是 React，项目中使用到的其他库也会执行生产环境版本的构建。但一定要注意，在开发过程中不要执行这项设置，因为它会让你失去很多重要的调试信息，并且代码的编译速度也会变慢。

2. 避免不必要的组件渲染

当组件的 props 或 state 发生变化时，组件的 render 方法会被重新调用，返回一个新的虚拟 DOM 对象。但在一些情况下，组件是没有必要重新调用 render 方法的。例如，父组件的每一次 render 调用都会触发子组件 componentWillReceiveProps 的调用，进而子组件的 render 方法也会被调用，但是这时候子组件的 props 可能并没有发生改变，改变的只是父组件的 props 或 state，所以这一次子组件的 render 是没有必要的，不仅多了一次 render 方法执行的时间，还多了一次虚拟 DOM 比较的时间。

React 组件的生命周期方法中提供了一个 shouldComponentUpdate 方法，这个方法的默认返回值是 true，如果返回 false，组件此次的更新将会停止，也就是后续的 componentWillUpdate、render 等方法都不会再被执行。我们可以把这个方法作为钩子，在这个方法中根据组件自身的业务逻辑决定返回 true 还是 false，从而避免组件不必要的渲染。例如，我们通过比较 props 中的一个自定义属性 item，决定是否需要继续组件的更新过程，代码如下：

```
class MyComponent extend React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    if(nextProps.item === this.props.item) {
      return false;
    }
    return true;
  }

  // ...
}
```

注意，示例中对 item 的比较是通过===比较对象的引用，所以即使两个对象的引用不相等，它们的内容也可能是相等的。最精确的比较方式是遍历对象的每一层级的属性分别比较，也就是进行深比较（deep compare），但 shouldComponentUpdate 被频繁调用，如果 props 和 state 的对象层级很深，深比较对性能的影响就比较大。一种折中的方案是，只比较对象的第一层级的属性，也就是执行浅比较（shallow compare）。例如下面两个对象：

```
const item = { foo, bar };
const nextItem = { foo, bar };
```

执行浅比较会使用===比较 item.foo 和 nextItem.foo、item.bar 和 nextItem.bar，而不会继续比较 foo、bar 的内容。React 中提供了一个 PureComponent 组件，这个组件会使用浅比较来比较新旧 props 和 state，因此可以通过让组件继承 PureComponent 来替代手写 shouldComponentUpdate 的逻辑。但是，使用浅比较很容易因为直接修改数据而产生错误，例如：

```
class NumberList extends React.PureComponent {
  constructor(props) {
```

```
    super(props);
    this.state = {
      numbers: [1,2,3,4]
    };
    this.handleClick = this.handleClick.bind(this);
  }
  // numbers 中新加一个数值
  handleClick() {
    const numbers = this.state.numbers;
    // 直接修改 numbers 对象
    numbers.push(numbers[numbers.length-1] + 1);
    this.setState({numbers: numbers});
  }

  render() {
    return (
      <div>
        <button onClick={this.handleClick} />
        {this.state.numbers.map(item => <div>{item}</div>)}
      </div>
    );
  }
}
```

点击 Button，NumberList 并不会重新调用 render，因为 handleClick 中是直接修改 this.state.numbers 这个数组的，this.state.numbers 的引用在 setState 前后并没有发生改变，所以 shouldComponentUpdate 会返回 false，从而终止组件的更新过程。在第 4 章深入理解组件 state 中，我们讲到要把 state 当作不可变对象，一个重要的原因就是为了提高组件 state 比较的效率。对于不可变对象来说，只需要比较对象的引用就能判断 state 是否发生改变。

3. 使用 key

在 5.2 节 Diff 算法中，我们已经解释了列表元素定义 key 的好处。React 会根据 key 索引元素，在 render 前后，拥有相同 key 值的元素是同一个元素，例如前面举过的例子：

```
// render 前
<ul>
  <li key="first" >first</li>
  <li key="second">second</li>
</ul>

// render 后
<ul>
  <li key="third">third</li>
```

```
  <li key="first">first</li>
  <li key="second">second</li>
</ul
```

定义 key 之后，React 并不会“傻瓜式”地按顺序依次更新每一个 li 元素：把第一个 li 元素更新为 third，把第二个 li 元素更新为 first，最后创建一个新的 li 元素，内容为 second。有了 key 的索引，React 知道 first 和 second 这两个 li 元素并没有发生变化，而只会在这两个 li 元素前面插入一个内容为 third 的 li 元素。可见，key 的使用减少了 DOM 操作，提高了 DOM 更新效率。当列表元素数量很多时，key 的使用更显得重要。

本章介绍的这三种性能优化方法是最常用的三种方法，其中使用生产环境版本的库是项目中必须采用的，使用 key 也推荐在项目中采用。通过重写 shouldComponentUpdate 方法避免不必要的组件渲染，这在项目开始阶段是可以不必在意的，大多数情况下，组件只是重复调用 render 方法对于性能的影响并不大。当发现项目确实存在性能问题时，再考虑通过这种方式进行优化也不迟。请大家记住，过早的优化并不是一件好事。

5.4 性能检测工具

我们可以通过一些性能检测工具更加方便地定位性能问题。

1. React Developer Tools for Chrome

这是一个 Chrome 插件，主要用来检测页面使用的 React 代码是否是生产环境版本。当访问网页时，如果插件图标的背景色是黑色的，就表示当前网页使用的是生产环境版本的 React，如图 5-1 所示。

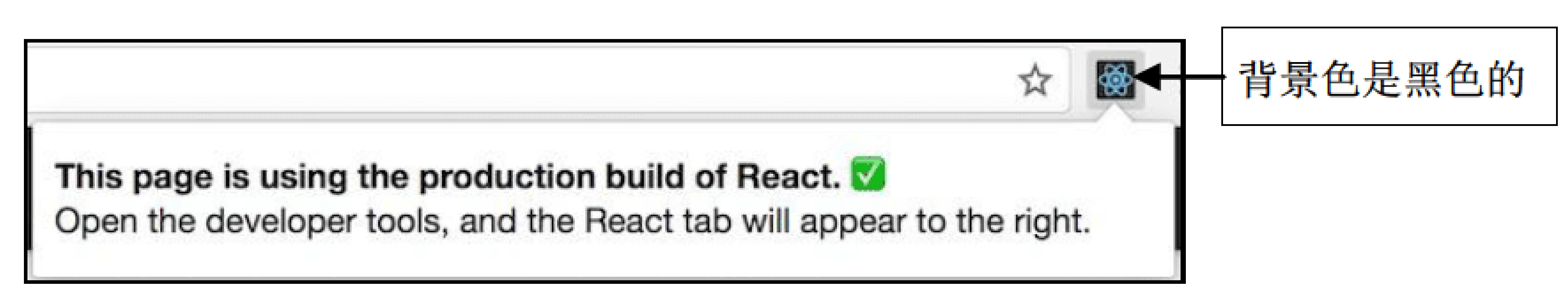


图 5-1

如果插件图标的背景色是红色的，就表示当前网页使用的是开发环境版本的 React，如图 5-2 所示。



图 5-2

2. Chrome Performance Tab

在开发模式下，可以通过 Chrome 浏览器提供的 Performance 工具观察组件的挂载、更新、卸载过程及各阶段使用的时间。使用方式为：

（1）确保应用运行在开发模式下。

（2）打开 Chrome 开发者工具，切换到 Performance 窗口，单击 Record 按钮开始统计。

（3）在页面上执行需要分析的操作，最好不要超过 20 秒，时间太长会导致 Chrome 卡死。

（4）单击 Stop 按钮结束统计，然后在 User Timing 里查看统计结果。

3. why-did-you-update

why-did-you-update 会比较组件的 state 和 props 的变化，从而发现组件 render 方法不必要的调用。

```
#安装
npm install why-did-you-update --save-dev

# 使用
import React from 'react'

if (process.env.NODE_ENV !== 'production') {
  const {whyDidYouUpdate} = require('why-did-you-update')
  whyDidYouUpdate(React)
}
```

5.5　本章小结

本章介绍了 React 的虚拟 DOM 机制以及用于虚拟 DOM 比较的 Diff 算法，虚拟 DOM 是 React 应用高效运行的基础。本章还介绍了常用的性能优化方法和性能检测工具，性能优化要避免过早优化问题。掌握本章的内容有助于开发出更加高效的 React 应用。

第 6 章

高阶组件

高阶组件是 React 中一个很重要且较复杂的概念，主要用来实现组件逻辑的抽象和复用，在很多第三方库（如 Redux）中都被使用到。即使开发一般的业务项目，如果能合理地使用高阶组件，也能显著提高项目的代码质量。本章将详细介绍高阶组件的概念及应用。

6.1 基本概念

在 JavaScript 中，高阶函数是以函数为参数，并且返回值也是函数的函数。类似地，高阶组件（简称 HOC）接收 React 组件作为参数，并且返回一个新的 React 组件。高阶组件本质上也是一个函数，并不是一个组件。高阶组件的函数形式如下：

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

我们先通过一个简单的例子看一下高阶组件是如何进行逻辑复用的。现在有一个组件 MyComponent，需要从 LocalStorage 中获取数据，然后渲染到界面。一般情况下，我们可以这样实现：

```
import React, { Component } from 'react'

class MyComponent extends Component {

  componentWillMount() {
    let data = localStorage.getItem('data');
    this.setState({data});
  }
```

```
  render() {
    return <div>{this.state.data}</div>
  }
}
```

代码很简单，但当其他组件也需要从 LocalStorage 中获取同样的数据展示出来时，每个组件都需要重写一次 componentWillMount 中的代码，这显然是很冗余的。下面让我们来看看使用高阶组件改写这部分代码。

```
import React, { Component } from 'react'

function withPersistentData(WrappedComponent) {
  return class extends Component {
    componentWillMount() {
      let data = localStorage.getItem('data');
       this.setState({data});
    }


    render() {
      // 通过{...this.props} 把传递给当前组件的属性继续传递给被包装的组件
      return <WrappedComponent data={this.state.data} {...this.props} />
    }
  }
}

class MyComponent extends Component {
  render() {
    return <div>{this.props.data}</div>
  }
}

const MyComponentWithPersistentData = withPersistentData(MyComponent)
```

withPersistentData 就是一个高阶组件，它返回一个新的组件，在新组件的 componentWillMount 中统一处理从 LocalStorage 中获取数据的逻辑，然后将获取到的数据通过 props 传递给被包装的组件 WrappedComponent，这样在 WrappedComponent 中就可以直接使用 this.props.data 获取需要展示的数据。当有其他的组件也需要这段逻辑时，继续使用 withPersistentData 这个高阶组件包装这些组件。

通过这个例子可以看出高阶组件的主要功能是封装并分离组件的通用逻辑，让通用逻辑在组件间更好地被复用。高阶组件的这种实现方式本质上是装饰者设计模式。

6.2　使用场景

高阶组件的使用场景主要有以下 4 种：

（1）操纵 props
（2）通过 ref 访问组件实例
（3）组件状态提升
（4）用其他元素包装组件

每一种使用场景通过一个例子来说明。

1. 操纵 props

在被包装组件接收 props 前，高阶组件可以先拦截到 props，对 props 执行增加、删除或修改的操作，然后将处理后的 props 再传递给被包装组件。6.1 节中的例子就属于这种情况，高阶组件为 WrappedComponent 增加了一个 data 属性。这里不再额外举例。

2. 通过 ref 访问组件实例

高阶组件通过 ref 获取被包装组件实例的引用，然后高阶组件就具备了直接操作被包装组件的属性或方法的能力。

```
function withRef(wrappedComponent) {
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.someMethod = this.someMethod.bind(this);
    }

    someMethod() {
      this.wrappedInstance.someMethodInWrappedComponent();
    }

    render() {
      //为被包装组件添加 ref 属性，从而获取该组件实例并赋值给 this.wrappedInstance
      return <WrappedComponent ref={(instance) => {this.wrappedInstance =
instance}} {...this.props} />
    }
  }
}
```

当 WrappedComponent 被渲染时，执行 ref 的回调函数，高阶组件通过 this.wrappedInstance 保存 WrappedComponent 实例的引用，在 someMethod 中，通过 this.wrappedInstance 调用 WrappedComponent 中的方法。这种用法在实际项目中很少会被用到，但当高阶组件封装的复用逻辑需要被包装组件的方法或属性的协同支持时，这种用法就有了用武之地。

3. 组件状态提升

在第 2 章中已经介绍过，无状态组件更容易被复用。高阶组件可以通过将被包装组件的状态及相应的状态处理方法提升到高阶组件自身内部实现被包装组件的无状态化。一个典型的场景是，

利用高阶组件将原本受控组件需要自己维护的状态统一提升到高阶组件中。

```
function withControlledState(WrappedComponent) {
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        value: ''
      };
      this.handleValueChange = this.handleValueChange.bind(this);
    }

    handleValueChange(event) {
      this.setState({
        value: event.target.value
      });
    }

    render() {
     // newProps 保存受控组件需要使用的属性和事件处理函数
      const newProps = {
        controlledProps: {
          value: this.state.value,
          onChange: this.handleValueChange
        }
      };
      return <WrappedComponent {...this.props} {...newProps}/>
    }
  }
}
```

这个例子把受控组件 value 属性用到的状态和处理 value 变化的回调函数都提升到高阶组件中，当我们再使用受控组件时，就可以这样使用：

```
class SimpleControlledComponent extends React.Component {
  render() {
    //此时的 SimpleControlledComponent 为无状态组件，状态由高阶组件维护
    return <input name="simple" {...this.props.controlledProps }/>
  }
}

const ComponentWithControlledState = withControlledState
(SimpleControlledComponent);
```

4. 用其他元素包装组件

我们还可以在高阶组件渲染 WrappedComponent 时添加额外的元素，这种情况通常用于为 WrappedComponent 增加布局或修改样式。

```
function withRedBackground(WrappedComponent) {
  return class extends React.Component {
    render() {
      return (
        <div style={{backgroundColor: 'red'}}>
          <WrappedComponent {...this.props}/>
        </div>
      )
    }
  }
}
```

6.3　参数传递

高阶组件的参数并非只能是一个组件，它还可以接收其他参数。例如，6.1 节的示例是从 LocalStorage 中获取 key 为 data 的数据，当需要获取的数据的 key 不确定时，withPersistentData 这个高阶组件就不满足需求了。我们可以让它接收一个额外的参数来决定从 LocalStorage 中获取哪个数据：

```
import React, { Component } from 'react'

function withPersistentData(WrappedComponent, key) {
  return class extends Component {
    componentWillMount() {
      let data = localStorage.getItem(key);
       this.setState({data});
    }

    render() {
      // 通过{...this.props} 把传递给当前组件的属性继续传递给被包装的组件
      return <WrappedComponent data={this.state.data} {...this.props} />
    }
  }
}

class MyComponent extends Component {
  render() {
    return <div>{this.props.data}</div>
  }
```

```
    }
    // 获取 key='data'的数据
    const MyComponent1WithPersistentData = withPersistentData(MyComponent,
'data');
    // 获取 key='name'的数据
    const MyComponent2WithPersistentData = withPersistentData(MyComponent,
'name');
```

新版本的 withPersistentData 满足获取不同 key 值的需求。但实际情况中，我们很少使用这种方式传递参数，而是采用更加灵活、更具通用性的函数形式：

```
HOC(...params)(WrappedComponent)
```

HOC(...params)的返回值是一个高阶组件，高阶组件需要的参数是先传递给 HOC 函数的。用这种形式改写 withPersistentData 如下（注意：这种形式的高阶组件使用箭头函数定义更为简洁）：

```
import React, { Component } from 'react'

function withPersistentData = (key) => (WrappedComponent) => {
  return class extends Component {
    componentWillMount() {
      let data = localStorage.getItem(key);
       this.setState({data});
    }

    render() {
      // 通过{...this.props} 把传递给当前组件的属性继续传递给被包装的组件
      return <WrappedComponent data={this.state.data} {...this.props} />
    }
  }
}

class MyComponent extends Component {
  render() {
    return <div>{this.props.data}</div>
  }
}

// 获取 key='data'的数据
const MyComponent1WithPersistentData = withPersistentData('data')
(MyComponent);
// 获取 key='name'的数据
const MyComponent2WithPersistentData = withPersistentData('name')
(MyComponent);
```

实际上，这种形式的高阶组件大量出现在第三方库中，例如 react-redux 中的 connect 函数就是一个典型的例子。connect 的简化定义如下：

```
connect(mapStateToProps, mapDispatchToProps)(WrappedComponent)
```

这个函数会将一个 React 组件连接到 Redux 的 store 上，在连接的过程中，connect 通过函数参数 mapStateToProps 从全局 store 中取出当前组件需要的 state，并把 state 转化成当前组件的 props；同时通过函数参数 mapDispatchToProps 把当前组件用到的 Redux 的 action creators 以 props 的方式传递给当前组件。connect 并不会修改传递进去的组件的定义，而是会返回一个新的组件。


注 意

connect 的参数 mapStateToProps、mapDispatchToProps 是函数类型，说明高阶组件的参数也可以是函数类型。

例如，把组件 ComponentA 连接到 Redux 上的写法类似于：

```
const ConnectedComponentA = connect(mapStateToProps, mapDispatchToProps)
(ComponentA);
```

我们可以把它拆分来看：

```
// connect 是一个函数，返回值 enhance 也是一个函数
const enhance = connect(mapStateToProps, mapDispatchToProps);
// enhance 是一个高阶组件
const ConnectedComponentA = enhance(ComponentA);
```

这种形式的高阶组件非常容易组合起来使用，因为当多个函数的输出和它的输入类型相同时，这些函数很容易组合到一起使用。例如，有 f、g、h 三个高阶组件，都只接收一个组件作为参数，于是我们可以很方便地嵌套使用它们：f(g(h(WrappedComponent)))。这里有一个例外，即最内层的高阶组件 h 可以有多个参数，但其他高阶组件必须只能接收一个参数，只有这样才能保证内层的函数返回值和外层的函数参数数量一致（都只有 1 个）。

例如，将 connect 和另一个打印日志的高阶组件 withLog()（注意，withLog()的执行结果才是真正的高阶组件）联合使用：

```
//connect 的参数是可选参数，这里省略了 mapDispatchToProps 参数
const ConnectedComponentA = connect(mapStateToProps)(withLog()
(ComponentA));
```

我们还可以定义一个工具函数 compose(...funcs)：

```
function compose(...funcs) {
  if (funcs.length === 0) {
    return arg => arg
  }

  if (funcs.length === 1) {
    return funcs[0]
  }

  return funcs.reduce((a,b) => (...args) => a(b(args)));
}
```

调用 compose(f, g, h)等价于 (...args) => f(g(h(...args)))。用 compose 函数可以把高阶组件嵌套的写法打平：

```
const enhance = compose(
  connect(mapStateToProps),
  withLog()
);
const ConnectedComponentA = enhance(ComponentA);
```

像 Redux 等很多第三方库都提供了 compose 的实现，compose 结合高阶组件使用可以显著提高代码的可读性和逻辑的清晰度。

6.4　继承方式实现高阶组件

前面介绍的高阶组件的实现方式都是由高阶组件处理通用逻辑，然后将相关属性传递给被包装组件，我们称这种实现方式为属性代理。除了属性代理外，还可以通过继承方式实现高阶组件：通过继承被包装组件实现逻辑的复用。继承方式实现的高阶组件常用于渲染劫持。例如，当用户处于登录状态时，允许组件渲染；否则渲染一个空组件。示例代码如下：

```
function withAuth(WrappedComponent) {
  return class extends WrappedComponent {
    render() {
      if (this.props.loggedIn) {
        return super.render();
      } else {
        return null;
      }
    }
  }
}
```

根据 WrappedComponent 的 this.props.loggedIn 判断用户是否已经登录，如果登录，就通过 super.render()调用 WrappedComponent 的 render 方法正常渲染组件，否则返回一个 null。继承方式实现的高阶组件对被包装组件具有侵入性，当组合多个高阶组件使用时，很容易因为子类组件忘记通过 super 调用父类组件方法而导致逻辑丢失。因此，在使用高阶组件时，应尽量通过代理方式实现高阶组件。

6.5　注意事项

使用高阶组件需要注意以下事项。

（1）为了在开发和调试阶段更好地区别包装了不同组件的高阶组件，需要对高阶组件的显示名称做自定义处理。常用的处理方式是，把被包装组件的显示名称也包到高阶组件的显示名称中。以 withPersistentData 为例：

```
function withPersistentData(WrappedComponent) {
  return class extends Component {
   //结合被包装组件的名称，自定义高阶组件的名称
    static displayName = `HOC(${getDisplayName(WrappedComponent)})`;

    render() {
      //...
    }
  }
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name ||
'Component';
}
```

（2）不要在组件的 render 方法中使用高阶组件，尽量也不要在组件的其他生命周期方法中使用高阶组件。因为调用高阶组件，每次都会返回一个新的组件，于是每次 render，前一次高阶组件创建的组件都会被卸载（unmount），然后重新挂载（mount）本次创建的新组件，既影响效率，又丢失了组件及其子组件的状态。例如：

```
render() {
  // 每次 render，enhance 都会创建一个新的组件，尽管被包装的组件没有变
  const EnhancedComponent = enhance(MyComponent);
  // 因为是新的组件，所以会经历旧组件的卸载和新组件的重新挂载
  return <EnhancedComponent />;
}
```

所以，高阶组件最适合使用的地方是在组件定义的外部，这样就不会受到组件生命周期的影响。

（3）如果需要使用被包装组件的静态方法，那么必须手动复制这些静态方法。因为高阶组件返回的新组件不包含被包装组件的静态方法。例如：

```
// WrappedComponent 组件定义了一个静态方法 staticMethod
WrappedComponent.staticMethod = function() {
  //...
}

function withHOC(WrappedComponent) {
  class Enhance extends React.Component {
    //...
  }
```

```
  // 手动复制静态方法到 Enhance 上
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

（4）Refs 不会被传递给被包装组件。尽管在定义高阶组件时，我们会把所有的属性都传递给被包装组件，但是 ref 并不会传递给被包装组件。如果在高阶组件的返回组件中定义了 ref，那么它指向的是这个返回的新组件，而不是内部被包装的组件。如果希望获取被包装组件的引用，那么可以自定义一个属性，属性的值是一个函数，传递给被包装组件的 ref 。下面的例子就是用 inputRef 这个属性名代替常规的 ref 命名：

```
function FocusInput({ inputRef, ...rest }) {
  // 使用高阶组件传递的 inputRef 作为 ref 的值
  return <input ref={inputRef} {...rest} />;
}

//enhance 是一个高阶组件
const EnhanceInput = enhance(FocusInput);

// 在一个组件的 render 方法中，自定义属性 inputRef 代替 ref，
// 保证 inputRef 可以传递给被包装组件
return (<EnhanceInput
  inputRef={(input) => {
    this.input = input
  }
}>)

// 组件内，让 FocusInput 自动获取焦点
this.input.focus();
```

（5）与父组件的区别。高阶组件在一些方面和父组件很相似。例如，我们完全可以把高阶组件中的逻辑放到一个父组件中去执行，执行完成的结果再传递给子组件，但是高阶组件强调的是逻辑的抽象。高阶组件是一个函数，函数关注的是逻辑；父组件是一个组件，组件主要关注的是 UI/DOM。如果逻辑是与 DOM 直接相关的，那么这部分逻辑适合放到父组件中实现；如果逻辑是与 DOM 不直接相关的，那么这部分逻辑适合使用高阶组件抽象，如数据校验、请求发送等。

6.6　本章小结

本章详细介绍了高阶组件。高阶组件主要用于封装组件的通用逻辑，常用在操纵组件 props、通过 ref 访问组件实例、组件状态提升和用其他元素包装组件等场景中。高阶组件可以接收被包装组件以外的其他参数，多个高阶组件还可以组合使用。高阶组件一般通过代理方式实现，少量场景中也会使用继承方式实现。灵活使用高阶组件可以显著提高代码质量。

第 3 篇　实战篇

在大型Web应用中使用React

第 7 章

路由：用 React Router 开发单页面应用

真实项目中，一般需要通过不同的 URL 标识不同的页面，也就是存在页面间路由的需求，这时候就该 React Router 发挥作用了。React Router 是 React 技术栈中最常用的用于构建单页面应用的解决方案。本章就来详细介绍 React Router。

7.1 基本用法

7.1.1 单页面应用和前端路由

在传统的 Web 应用中，浏览器根据地址栏的 URL 向服务器发送一个 HTTP 请求，服务器根据 URL 返回一个 HTML 页面。这种情况下，一个 URL 对应一个 HTML 页面，一个 Web 应用包含很多 HTML 页面，这样的应用就是多页面应用；在多页面应用中，页面路由的控制由服务器端负责，这种路由方式称为后端路由。

在多页面应用中，每次页面切换都需要向服务器发送一次请求，页面使用到的静态资源也需要重新请求加载，存在一定的浪费。而且，页面的整体刷新对用户体验也有影响，因为不同页面间往往存在共同的部分，例如导航栏、侧边栏等，页面整体刷新也会导致共用部分的刷新。

有没有一种方式让 Web 应用只是看起来像多页面应用，实际 URL 的变化可以引起页面内容的变化，但不会向服务器发送新的请求呢？实际上，满足这种要求的 Web 应用就是单页面应用（Single Page Application，简称 SPA）。单页面应用虽然名为“单页”，但视觉上的感受仍然是多页面，因为 URL 发生变化，页面的内容也会发生变化，但这只是逻辑上的多页面，实际上无论 URL 如何变化，对应的 HTML 文件都是同一个，这也是单页面应用名字的由来。在单页面应用中，URL 发生

变化并不会向服务器发送新的请求，所以“逻辑页面”（这个名称用来和真实的 HTML 页面区分）的路由只能由前端负责，这种路由方式称为前端路由。

React Router 就是一种前端路由的实现方式。通过使用 React Router 可以让 Web 应用根据不同的 URL 渲染不同的组件，这样的组件渲染方式可以解决更加复杂的业务场景。例如，当 URL 的 pathname 为/list 时，页面会渲染一个列表组件，当点击列表中的一项时，pathname 更改为/item/:id（id 为参数），旧的列表组件会被卸载，取而代之的是一个新的单一项的详情组件。



目前，国内的搜索引擎大多对单页面应用的 SEO 支持的不好，因此，对于 SEO 非常看重的 Web 应用（例如，企业官方网站、电商网站等），一般还是会选择采用多页面应用。React 也并非只能用于开发单页面应用。

7.1.2　React Router 的安装

本书使用的 React Router 的大版本号是 v4，这也是写作本书时的最新版本。

React Router 包含 3 个库：react-router、react-router-dom 和 react-router-native。react-router 提供最基本的路由功能，实际使用时，我们不会直接安装 react-router，而是根据应用运行的环境选择安装 react-router-dom（在浏览器中使用）或 react-router-native（在 react-native 中使用）。react-router-dom 和 react-router-native 都依赖于 react-router，所以在安装时，react-router 也会自动安装。因为本书介绍的是创建 Web 应用，所以这里需要安装 react-router-dom：

```
npm install react-router-dom
```



React Router v4 是对 React Router 的一次彻底重构，采用动态路由，遵循 React 中一切皆组件的思想，每一个 Route（路由）都是一个普通的 React 组件，这一点也导致 v4 版本较之前的版本在 API 和使用方式上都有了巨大变化，也就是说，v4 版本并不兼容之前的 React Router 版本，请读者务必注意。

7.1.3　路由器

React Router 通过 Router 和 Route 两个组件完成路由功能。Router 可以理解成路由器，一个应用中只需要一个 Router 实例，所有的路由配置组件 Route 都定义为 Router 的子组件。在 Web 应用中，我们一般会使用对 Router 进行包装的 BrowserRouter 或 HashRouter 两个组件。BrowserRouter 使用 HTML 5 的 history API（pushState、replaceState 等）实现应用的 UI 和 URL 的同步。HashRouter 使用 URL 的 hash 实现应用的 UI 和 URL 的同步。

BrowserRouter 创建的 URL 形式如下：

http://example.com/some/path

HashRouter 创建的 URL 形式如下：

http://example.com/#/some/path

使用 BrowserRouter 时，一般还需要对服务器进行配置，让服务器能正确地处理所有可能的 URL。例如，当浏览器发送 http://example.com/some/path 和 http://example.com/some/path2 两个请求

时，服务器需要能返回正确的 HTML 页面（也就是单页面应用中唯一的 HTML 页面）。使用 HashRouter 则不存在这个问题，因为 hash 部分的内容会被服务器自动忽略，真正有效的信息是 hash 前面的部分，而对于单页面应用来说，这部分内容是固定的。

Router 会创建一个 history 对象，history 用来跟踪 URL，当 URL 发生变化时，Router 的后代组件会重新渲染。React Router 中提供的其他组件可以通过 context 获取 history 对象（在 4.3 节组件通信中已经详细介绍过 context），这也隐含说明了 React Router 中的其他组件必须作为 Router 组件的后代组件使用。但 Router 中只能有唯一的一个子元素，例如：

```
// 正确
ReactDOM.render((
  <BrowserRouter>
    <App />
  </BrowserRouter>
), document.getElementById('root'))

// 错误，Router 中包含两个子元素
ReactDOM.render((
  <BrowserRouter>
    <App1 />
    <App2 />
  </BrowserRouter>
), document.getElementById('root'))
```

7.1.4　路由配置

Route 是 React Router 中用于配置路由信息的组件，也是 React Router 中使用频率最高的组件。每当有一个组件需要根据 URL 决定是否渲染时，就需要创建一个 Route。

1. path

每个 Route 都需要定义一个 path 属性，当使用 BrowserRouter 时，path 用来描述这个 Route 匹配的 URL 的 pathname；当使用 HashRouter 时，path 用来描述这个 Route 匹配的 URL 的 hash。例如，使用 BrowserRouter 时，<Route path='/foo' />会匹配一个 pathname 以 foo 开始的 URL（如 http://example.com/foo）。当 URL 匹配一个 Route 时，这个 Route 中定义的组件就会被渲染出来；反之，Route 不进行渲染（Route 使用 children 属性渲染组件除外，可参考下文）。


注　意

本章中的示例，如无特殊说明，使用的都是 BrowserRouter。

2. match

当 URL 和 Route 匹配时，Route 会创建一个 match 对象作为 props 中的一个属性传递给被渲染的组件。这个对象包含以下 4 个属性。

（1）params：Route 的 path 可以包含参数，例如<Route path='/foo/:id'>包含一个参数 id。params 就是用于从匹配的 URL 中解析出 path 中的参数，例如，当 URL="http://example.com/foo/1"时，params = {id: 1}。

（2）isExact：是一个布尔值，当 URL 完全匹配时，值为 true；当 URL 部分匹配时，值为 false。例如，当 path="/foo"、URL="http://example.com/foo"时，是完全匹配；当 URL="http://example.com/foo/1"时，是部分匹配。

（3）path：Route 的 path 属性，构建嵌套路由时会使用到。

（4）url：URL 的匹配部分。

3. Route 渲染组件的方式

Route 如何决定渲染的内容呢？Route 提供了 3 个属性，用于定义待渲染的组件：

■ component

component 的值是一个组件，当 URL 和 Route 匹配时，component 属性定义的组件就会被渲染。例如：

```
<Route path='/foo' component={Foo}>
```

当 URL="http://example.com/foo"时，Foo 组件会被渲染。

■ render

render 的值是一个函数，这个函数返回一个 React 元素。这种方式可以方便地为待渲染的组件传递额外的属性。例如：

```
<Route path='/foo' render={(props) => (
  <Foo {...props} data={extraProps} />
)}>
```

Foo 组件接收了一个额外的 data 属性。

■ children

children 的值也是一个函数，函数返回要渲染的 React 元素。与前两种方式不同之处是，无论是否匹配成功，children 返回的组件都会被渲染。但是，当匹配不成功时，match 属性为 null。例如：

```
<Route path='/foo' children={(props) => (
  <div className={props.match ? 'active' : ''}>
    <Foo />
  </div>
</Route>
```

如果 Route 匹配当前 URL，待渲染元素的根节点 div 的 class 将被设置成 active。

4. Switch 和 exact

当 URL 和多个 Route 匹配时，这些 Route 都会执行渲染操作。如果只想让第一个匹配的 Route 渲染，那么可以把这些 Route 包到一个 Switch 组件中。如果想让 URL 和 Route 完全匹配时，Route 才渲染，那么可以使用 Route 的 exact 属性。Switch 和 exact 常常联合使用，用于应用首页的导航。例如：

```
<Router>
  <Switch>
```

```
      <Route exact path='/' component={Home}/>
      <Route path='/posts' component={Posts}/>
      <Route path='/:user' component={User}/>
    </Switch>
  </Router>
```

如果不使用 Switch，当 URL 的 pathname 为"/posts"时，<Route path='/posts' />和<Route path='/:user' />都会被匹配，但显然我们并不希望<Route path='/:user' />被匹配，实际上也没有用户名为 posts 的用户。如果不使用 exact，"/" "/posts" "/user1"等几乎所有 URL 都会匹配第一个 Route，又因为 Switch 的存在，后面的两个 Route 永远也不会被匹配。使用 exact，保证只有当 URL 的 pathname 为"/"时，第一个 Route 才会被匹配。

5. 嵌套路由

嵌套路由是指在 Route 渲染的组件内部定义新的 Route。例如，在上一个例子中，在 Posts 组件内再定义两个 Route:

```
const Posts = ({ match }) => {
  return (
    <div>
      {/* 这里 match.url 等于 /posts */}
      <Route path={`${match.url}/:id`} component={PostDetail} />
      <Route exact path={match.url} component={PostList} />
    </div>
  );
}
```

当 URL 的 pathname 为"/posts/react"时，PostDetail 组件会被渲染；当 URL 的 pathname 为"/posts"时，PostList 组件会被渲染。Route 的嵌套使用让应用可以更加灵活地使用路由。

7.1.5　链接

Link 是 React Router 提供的链接组件，一个 Link 组件定义了当点击该 Link 时，页面应该如何路由。例如：

```
const Navigation = () => (
  <header>
    <nav>
      <ul>
        <li><Link to='/'>Home</Link></li>
        <li><Link to='/posts'>Posts</Link></li>
      </ul>
    </nav>
  </header>
)
```

Link 使用 to 属性声明要导航到的 URL 地址。to 可以是 string 或 object 类型，当 to 为 object 类型时，可以包含 pathname、search、hash、state 四个属性，例如：

```
<Link to={{
  pathname: '/posts',
  search: '?sort=name',
  hash: '#the-hash',
  state: { fromHome: true }
}}/>
```

除了使用 Link 外，我们还可以使用 history 对象手动实现导航。history 中最常用的两个方法是 push(path, [state])和 replace(path, [state])，push 会向浏览历史记录中新增一条记录，replace 会用新记录替换当前记录。例如：

```
history.push('/posts')
history.replace('/posts')
```

7.2 项目实战

本节将使用 React Router 继续完善 BBS 项目。为了避免由于项目过于复杂而不便讲解和理解，因此这里只抽象出 BBS 项目中最核心的三个页面：

- 登录页，负责应用的登录功能。
- 帖子列表页，以列表形式展示所有帖子的基本信息。
- 帖子详情页，展示某个帖子的详细内容。

具备这三个页面，BBS 的核心功能也就基本完整了。本节项目源代码的目录为/chapter-07/bbs-router。

7.2.1 后台服务 API 介绍

真实 BBS 项目的数据肯定要保存到后台的数据库中，同时需要依赖后台服务提供的 API 实现对应用所需数据的增、删、改、查操作。本书不涉及后台开发的内容，为了更接近真实的项目开发场景，使用 APICloud 的数据云功能快速地生成了所需的 API。API 的生成方式不做介绍，这里主要介绍 API 的使用。

按照业务功能划分，API 可以分为三类：登录、帖子和评论。

1. 登录：/user/login

执行用户登录验证。注销不调用后台的 API，只在客户端清除登录信息。注意，这是一个简化的登录和注销流程，并不适用于真实项目。



因为示例项目不涉及注册功能，我们预先在后台创建好三个账号，用户名分别是：tom、jack、steve，密码均是：123456。读者可使用这三个账号登录应用。

2. 帖子：/post

与帖子相关的操作都通过这个 API 完成，包括获取帖子列表数据、获取某个帖子的详情、新增帖子、修改帖子等。APICloud 生成的 API 是 RESTful API，因此/post 实际上相当于多个 API，例如以 HTTP Get 方法调用接口用于获取帖子数据，以 HTTP Post 方法调用接口用于新增帖子，以 Http Put 方法调用接口用于修改帖子。这属于 RESTful API 规范，这里不再展开。

3. 评论：/comment

与评论相关的操作都通过这个 API 完成，包括获取某一个帖子的评论列表和新增一条评论。为方便调用，API 相关信息已被封装到 utils/url.js 中，代码如下：

```
// 获取帖子列表的过滤条件
const postListFilter = {
  fields: ["id", "title", "author", "vote", "updatedAt"],
  limit: 10,
  order: "updatedAt DESC",
  include: "authorPointer",
  includefilter: { user: { fields: ["id", "username"] } }
};

// 获取帖子详情的过滤条件
const postByIdFilter = id => ({
  fields: ["id", "title", "author", "vote", "updatedAt", "content"],
  where: { id: id },
  include: "authorPointer",
  includefilter: { user: { fields: ["id", "username"] } }
});

// 获取评论列表的过滤条件
const commentListFilter = postId => ({
  fields: ["id", "author", "updatedAt", "content"],
  where: { post: postId },
  limit: 20,
  order: "updatedAt DESC",
  include: "authorPointer",
  includefilter: { user: { fields: ["id", "username"] } }
});

function encodeFilter(filter) {
  return encodeURIComponent(JSON.stringify(filter));
}

export default {
```

```
  // 登录
  login: () => "/user/login",
  // 获取帖子列表
  getPostList: () => `/post?filter=${encodeFilter(postListFilter)}`,
  // 获取帖子详情
  getPostById: id => `/post?filter=${encodeFilter(postByIdFilter(id))}`,
  // 新建帖子
  createPost: () => "/post",
  // 修改帖子
  updatePost: id => `/post/${id}`,
  // 获取评论列表
  getCommentList: postId =>
    `/comment?filter=${encodeFilter(commentListFilter(postId))}`,
  // 新建评论
  createComment: () => "/comment"
};
```

读者重点关注最后 export 的对象，这个对象包含项目中需要使用的所有 API 信息。至于前面定义的变量和函数，是根据 APICloud 的规范提供 API 调用时所需的过滤条件，可不必深究。

我们使用 HTML 5 fetch 接口调用 API，在 utils/request.js 中对 fetch 进行了封装，定义了 get、post、put 三个方法，分别满足以不同 HTTP 方法（Get、Post、Put）调用 API 的场景。主要代码如下：

```
function get(url) {
  return fetch(url, {
    method: "GET",
    headers: headers,
  }).then(response => {
    return handleResponse(url, response);
  }).catch(err => {
    console.error(`Request failed. Url = ${url} . Message = ${err}`);
    return {error: {message: "Request failed."}};
  })
}

function post(url, data) {
  return fetch(url, {
    method: "POST",
    headers: headers,
    body: JSON.stringify(data)
  }).then(response => {
    return handleResponse(url, response);
  }).catch(err => {
    console.error(`Request failed. Url = ${url} . Message = ${err}`);
    return {error: {message: "Request failed."}};
```

```
  })
}

function put(url, data) {
  return fetch(url, {
    method: "PUT",
    headers: headers,
    body: JSON.stringify(data)
  }).then(response => {
    return handleResponse(url, response);
  }).catch(err => {
    console.error(`Request failed. Url = ${url} . Message = ${err}`);
    return {error: {message: "Request failed."}};
  })
}

function handleResponse(url, response) {
  if(response.status < 500){
    return response.json();
  }else{
    console.error(`Request failed. Url = ${url} . Message =
${response.statusText}`);
    return {error: {message: "Request failed due to server error "}};
  }
}

export {get, post, put}
```

因为 APICloud 提供的 API 和本地程序运行在不同域下，所以本地程序直接调用 APICloud 的 API 会存在跨域调用的问题。我们利用代理服务器解决这个问题。在 create-react-app 中使用代理很简单，只需要在项目的 package.json 中配置 proxy 属性，proxy 的值是请求要转发到的最终地址。APICloud 提供的 API 运行在 https://d.apicloud.com/mcm/api 下，因此配置如下：

```
"proxy": "https://d.apicloud.com/mcm/api"
```

但需要注意，使用这种方式配置代理只在开发环境模式下有效，即 npm start 启动程序时，代理有效。

7.2.2　路由设计

路由设计的过程可以分为两步：

（1）为每一个页面定义有语义的路由名称（path）。

（2）组织 Route 结构层次。

1. 定义路由名称

我们有三个页面，按照页面功能不难定义出如下的路由名称：

- 登录页：/login。
- 帖子列表页：/posts。
- 帖子详情页：/posts/:id（id 代表帖子的 ID）。

但是这些还不够，还需要考虑打开应用时的默认页面，也就是根路径"/"对应的页面。结合业务场景，帖子列表页作为应用的默认页面最为合适，因此，帖子列表页对应两个路由名称："/posts"和"/"。

2. 组织 Route 结构层次

React Router 4 并不需要在一个地方集中声明应用需要的所有 Route，Route 实际上也是一个普通的 React 组件，可以在任意地方使用它（前提是，Route 必须是 Router 的子节点）。当然，这样的灵活性也一定程度上增加了组织 Route 结构层次的难度。

我们先来考虑第一层级的路由。登录页和帖子列表页（首页）应该属于第一层级：

```
<Router>
  <Switch>
    <Route exact path="/" component={Home} />
    <Route path="/login" component={Login} />
    <Route path="/posts" component={Home} />
  </Switch>
</Router>
```

第一个 Route 使用了 exact 属性，保证只有当访问根路径时，第一个 Route 才会匹配成功。Home 是首页对应的组件，可以通过"/posts"和"/"两个路径访问首页。注意，这里并没有直接渲染帖子列表组件，真正渲染帖子列表组件的地方在 Home 组件内，通过第二层级的路由处理帖子列表组件和帖子详情组件的渲染，components/Home.js 的主要代码如下：

```
class Home extends Component {
  /** 省略其余代码 **/

  render() {
    const { match, location } = this.props;
    const { username } = this.state;
    return (
      <div>
        <Header
          username={username}
          onLogout={this.handleLogout}
          location={location}
        />
        {/* 帖子列表路由配置 */}
```

```
          <Route
            path={match.url}
            exact
            render={props => <PostList username={username} {...props} />}
          />
          {/* 帖子详情路由配置 */}
          <Route
            path={`${match.url}/:id`}
            render={props => <Post username={username} {...props} />}
          />
        </div>
      );
    }
  }
```

Home 的 render 内定义了两个 Route，分别用于渲染帖子列表和帖子详情。PostList 是帖子列表组件，Post 是帖子详情组件，代码使用 Route 的 render 属性渲染这两个组件，因为它们需要接收额外的 username 属性。另外，无论访问的是帖子列表页面（首页）还是帖子详情页面，都会共用相同的 Header 组件。

7.2.3　登录页

登录页的界面很简单，只需要提供一个 form 表单供用户输入登录信息即可。因此，components/Login.js 中的 render 方法如下：

```
render() {
  // from 保存跳转到登录页前的页面路径，用于在登录成功后重定向到原来的页面
  const { from } = this.props.location.state || { from: { pathname: "/" } };
  const { redirectToReferrer } = this.state;
  // 登录成功后，redirectToReferrer 为 true，使用 Redirect 组件重定向页面
  if (redirectToReferrer) {
    return <Redirect to={from} />;
  }
  return (
    <form className="login" onSubmit={this.handleSubmit}>
      <div>
        <label>
          用户名：
          <input
            name="username"
            type="text"
            value={this.state.username}
            onChange={this.handleChange}
          />
```

```
          </label>
        </div>
        <div>
          <label>
            密码：
            <input
              name="password"
              type="password"
              value={this.state.password}
              onChange={this.handleChange}
            />
          </label>
        </div>
        <input type="submit" value="登录" />
      </form>
    );
  }
```

当用户点击“登录”按钮时，会调用后台的登录 API 进行验证，登录成功后，将用户信息存储到 sessionStorage 中，其他页面根据 sessionStorage 中是否有用户信息判断应用是否处于登录状态。登录逻辑代码如下：

```
handleSubmit(e) {
  e.preventDefault();
  const username = this.state.username;
  const password = this.state.password;
  if (username.length === 0 || password.length === 0) {
    alert("用户名或密码不能为空！");
    return;
  }
  const params = {
    username,
    password
  };
  post(url.login(), params).then(data => {
    if (data.error) {
      alert(data.error.message || "login failed");
    } else {
      // 保存登录信息到 sessionStorage
      sessionStorage.setItem("userId", data.userId);
      sessionStorage.setItem("username", username);
      // 登录成功后，设置 redirectToReferrer 为 true
      this.setState({
```

```
        redirectToReferrer: true
      });
    }
  });
}
```

登录成功后，Login 组件会修改 state 中的 redirectToReferrer 为 true，当再次 render 时，会渲染 React Router 的 Redirect 组件，这个组件用于页面的重定向，将页面重定向到登录前的页面或首页（没有上一个页面的情况下）。完整的 components/Login.js 的代码如下：

```
import React, { Component } from "react";
import { Redirect } from "react-router-dom";
import { post } from "../utils/request";
import url from "../utils/url";
import "./Login.css";

class Login extends Component {
  constructor(props) {
    super(props);
    this.state = {
      username: "",
      password: "",
      redirectToReferrer: false   // 是否重定向到之前的页面
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  // 处理用户名、密码的变化
  handleChange(e) {
    if (e.target.name === "username") {
      this.setState({
        username: e.target.value
      });
    } else if (e.target.name === "password") {
      this.setState({
        password: e.target.value
      });
    } else {
      // do nothing
    }
  }

  // 提交登录表单
  handleSubmit(e) {
    e.preventDefault();
```

```
    const username = this.state.username;
    const password = this.state.password;
    if (username.length === 0 || password.length === 0) {
      alert("用户名或密码不能为空！");
      return;
    }
    const params = {
      username,
      password
    };
    post(url.login(), params).then(data => {
      if (data.error) {
        alert(data.error.message || "login failed");
      } else {
        // 保存登录信息到 sessionStorage
        sessionStorage.setItem("userId", data.userId);
        sessionStorage.setItem("username", username);
        // 登录成功后，设置 redirectToReferrer 为 true
        this.setState({
          redirectToReferrer: true
        });
      }
    });
  }

  render() {
    // from 保存跳转到登录页前的页面路径，用于在登录成功后重定向到原来的页面
    const { from } = this.props.location.state || { from: { pathname: "/" } };
    const { redirectToReferrer } = this.state;
    // 登录成功后，redirectToReferrer 为 true，使用 Redirect 组件重定向页面
    if (redirectToReferrer) {
      return <Redirect to={from} />;
    }
    return (
      <form className="login" onSubmit={this.handleSubmit}>
        <div>
          <label>
            用户名:
            <input
              name="username"
              type="text"
              value={this.state.username}
              onChange={this.handleChange}
```

```
            />
          </label>
        </div>
        <div>
          <label>
            密码：
            <input
              name="password"
              type="password"
              value={this.state.password}
              onChange={this.handleChange}
            />
          </label>
        </div>
        <input type="submit" value="登录" />
      </form>
    );
  }
}

export default Login;
```

7.2.4　帖子列表页

帖子列表页也是项目的首页。我们先划分出页面中的主要组件，如图 7-1 所示。



图 7-1

页面划分为 Header 和 PostList 两大组件，PostList 中又包含 PostEditor 和 PostsView 组件，PostsView 中包含 PostItem 组件。这只是其中一种组件划分方式，组件划分的方式并不唯一，还可以使用其他划分方式，例如把 PostEditor 移到 PostList 外，让它和 PostList 处于同一层级。如果帖子列表的每一项样式和逻辑都很简单，就不需要单独拆分出 PostItem 组件。总之，划分页面组件需要根据页面的结构、组件的复用性、组件的复杂度等因素综合考虑。


注 意

划分组件容易走两个极端，组件粒度过大和组件粒度过小。若组件粒度过大，则组件逻辑过于复杂，组件的可维护性和复用性都变差；若组件粒度过小，则项目中的组件数量激增，一个简单功能往往需要引入大量组件，增加开发成本，组件数量过多也不便于查找。一种观点是，一个组件只负责一个功能。对于这种观点，建议大家辩证地看待，如果几个功能都很简单，且每一个功能都没有复用的需求，那么将这几个功能放到一个组件中也未尝不可，这样做可以提高开发效率。事实上，本书中有些组件也是包含多个简单功能的。

下面我们就来逐一分析这几个组件。

1. Header

Header 组件定义了页面的顶部导航栏，其中使用了两个 Link 组件，分别导航到首页和登录页。代码如下：

```
import React, { Component } from "react";
import { Link } from "react-router-dom";
import "./Header.css";

class Header extends Component {
  render() {
    const { username, onLogout, location } = this.props;
    return (
      <div className="header">
        <div className="nav">
          <span className="left-link">
            <Link to="/">首页</Link>
          </span>
          {/* 用户已登录，显示登录用户的信息；否则显示登录按钮 */}
          {username && username.length > 0 ? (
            <span className="user">
              当前用户：{username} 
              <button onClick={onLogout}>注销</button>
            </span>
          ) : (
            <span className="right-link">
              {/* 通过 state 属性，保存当前页面的地址 */}
              <Link to={{ pathname: "/login", state: { from: location } }}>
```

```
                登录
              </Link>
            </span>
          )}
        </div>
      </div>
    );
  }
}

export default Header;
```

导航到登录页 Link 的 to 属性的值不是一个字符串，而是一个对象：{ pathname: "/login", state: { from: location } }。对象中的 location 是当前页面的位置，这样在 Login 组件执行完登录逻辑后，可以从 this.props.location.state 中获取上一个页面的 location，然后重定向到上一个页面。

2. PostList

PostList 是这个页面中最复杂的组件，它负责获取帖子列表数据、保存新建的帖子以及控制 PostEditor 的显示与隐藏。PostList 的 state 包含两个属性，即 posts 和 newPost，分别用于保存帖子列表数据和判断当前是否正在创建新的帖子。

当 PostList 组件挂载后，调用后台 API 获取列表数据，代码如下：

```
componentDidMount() {
  this.refreshPostList();
}

refreshPostList() {
  // 调用后台 API 获取列表数据，并将返回的数据设置到 state 中
  get(url.getPostList()).then(data => {
    if (!data.error) {
      this.setState({
        posts: data,
        newPost: false
      });
    }
  });
}
```

PostList 的 render 方法如下：

```
render() {
  const { userId } = this.props;
  return (
    <div className="postList">
      <div>
```

```
          <h2>帖子列表</h2>
          {/* 只有在登录状态，才显示发帖按钮 */}
          {userId ? <button onClick={this.handleNewPost}>发帖</button> : null}
        </div>
          {/* 若当前正在创建新帖子，则渲染 PostEditor 组件 */}
        {this.state.newPost ? (
          <PostEditor onSave={this.handleSave} onCancel={this.handleCancel} />
        ) : null}
        {/* PostsView 显示帖子的列表数据 */}
        <PostsView posts={this.state.posts} />
      </div>
    );
  }
```

render 中渲染组件 PostEditor 和 PostsView，PostsView 用于展示帖子列表，PostEditor 用于创建新的帖子。保存新帖子的回调函数 handleSave 也定义在 PostList 中，主要执行的逻辑是调用后台 API 保存新帖子，保存完成后，再刷新帖子列表数据，代码如下：

```
  handleSave(data) {
    // 当前登录用户的信息和默认的点赞数，同帖子的标题和内容，共同构成最终待保存的帖子对象
    const postData = { ...data, author: this.props.userId, vote: 0 };
    post(url.createPost(), postData).then(data => {
      if (!data.error) {
        // 保存成功后，刷新帖子列表
        this.refreshPostList();
      }
    });
  }
```

另外，PostList 控制 PostEditor 的显示与隐藏的逻辑是：当用户处于登录状态且点击了发帖按钮后，PostEditor 会被渲染。

components/PostList.js 的完整代码如下：

```
import React, { Component } from "react";
import { Link } from "react-router-dom";
import PostsView from "./PostsView";
import PostEditor from "./PostEditor";
import { get, post } from "../utils/request";
import url from "../utils/url";
import "./PostList.css";

class PostList extends Component {
  constructor(props) {
    super(props);
    this.state = {
```

```
    posts: [],
    newPost: false
  };
  this.handleCancel = this.handleCancel.bind(this);
  this.handleSave = this.handleSave.bind(this);
  this.handleNewPost = this.handleNewPost.bind(this);
  this.refreshPostList = this.refreshPostList.bind(this);
}

componentDidMount() {
  this.refreshPostList();
}

// 获取帖子列表
refreshPostList() {
  // 调用后台 API 获取列表数据，并将返回的数据设置到 state 中
  get(url.getPostList()).then(data => {
    if (!data.error) {
      this.setState({
        posts: data,
        newPost: false
      });
    }
  });
}

// 保存帖子
handleSave(data) {
  // 当前登录用户的信息和默认的点赞数，同帖子的标题和内容，共同构成最终待保存的帖子对象
  const postData = { ...data, author: this.props.userId, vote: 0 };
  post(url.createPost(), postData).then(data => {
    if (!data.error) {
      // 保存成功后，刷新帖子列表
      this.refreshPostList();
    }
  });
}

// 取消新建帖子
handleCancel() {
  this.setState({
    newPost: false
  });
}
```

```
  // 新建帖子
  handleNewPost() {
    this.setState({
      newPost: true
    });
  }

  render() {
    const { userId } = this.props;
    return (
      <div className="postList">
        <div>
          <h2>帖子列表</h2>
          {/* 只有在登录状态，才显示发帖按钮 */}
          {userId ? <button onClick={this.handleNewPost}>发帖</button> : null}
        </div>
        {/* 若当前正在创建新帖子，则渲染 PostEditor 组件 */}
        {this.state.newPost ? (
          <PostEditor onSave={this.handleSave} onCancel={this.handleCancel} />
        ) : null}
        {/* PostsView 显示帖子的列表数据 */}
        <PostsView posts={this.state.posts} />
      </div>
    );
  }
}

export default PostList;
```

3. PostEditor

PostEditor 用于编辑帖子的信息，不仅会在帖子列表页中使用，在帖子详情页中也会使用。在帖子列表页，PostEditor 用于发布新帖子；在帖子详情页，PostEditor 用于修改当前帖子的信息。PostEditor 的 UI 很简单，主要由一个 input 和一个 textarea 组成，负责输入帖子的标题和正文。PostEditor 只负责界面逻辑，真正保存数据的逻辑是通过调用父组件的回调函数来完成的，components/PostEditor.js 的代码如下：

```
import React, { Component } from "react";
import "./PostEditor.css";

class PostEditor extends Component {
  constructor(props) {
    super(props);
    const { post } = this.props;
    this.state = {
```

```
      title: (post && post.title) || "",
      content: (post && post.content) || ""
    };
    this.handleCancelClick = this.handleCancelClick.bind(this);
    this.handleSaveClick = this.handleSaveClick.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }

  // 处理帖子的编辑信息
  handleChange(e) {
    const name = e.target.name;
    if (name === "title") {
      this.setState({
        title: e.target.value
      });
    } else if (name === "content") {
      this.setState({
        content: e.target.value
      });
    } else {
    }
  }

  // 取消帖子的编辑
  handleCancelClick() {
    this.props.onCancel();
  }

  // 保存帖子
  handleSaveClick() {
    const data = {
      title: this.state.title,
      content: this.state.content
    };
    // 调用父组件的回调函数执行真正的保存逻辑
    this.props.onSave(data);
  }

  render() {
    return (
      <div className="postEditor">
        <input
          type="text"
          name="title"
          placeholder="标题"
```

```
          value={this.state.title}
          onChange={this.handleChange}
        />
        <textarea
          name="content"
          placeholder="内容"
          value={this.state.content}
          onChange={this.handleChange}
        />
        <button onClick={this.handleCancelClick}>取消</button>
        <button onClick={this.handleSaveClick}>保存</button>
      </div>
    );
  }
}

export default PostEditor;
```

4. PostsView

PostsView 负责显示帖子列表。PostsView 渲染 PostItem 时，每个 PostItem 外面都包裹了一个 React Router 的 Link 组件，这样点击每一个帖子项，都会跳转到该帖子的详情页。components/PostsView.js 的代码如下：

```
import React, { Component } from 'react';
import { Link } from "react-router-dom";
import PostItem from "./PostItem";

class PostsView extends Component {
  render() {
    const { posts } = this.props
    return (
      <ul>
        {posts.map(item => (
          // 使用 Link 组件包裹每一个 PostItem
          <Link key={item.id} to={`/posts/${item.id}`}>
            <PostItem post={item} />
          </Link>
        ))}
      </ul>
    );
  }
}

export default PostsView;
```

5. PostItem

PostItem 组件用于渲染帖子列表的每一项，它不负责任何业务逻辑，只关注组件的渲染，我们使用一个无状态的函数组件实现 PostItem。components/PostItem.js 的代码如下：

```
import React from "react";
import { getFormatDate } from "../utils/date";
import "./PostItem.css";
import like from "../images/like.png";

function PostItem(props) {
  const { post } = props;
  return (
    <li className="postItem">
      <div className="title">{post.title}</div>
      <div>
        创建人：<span>{post.author.username}</span>
      </div>
      <div>
        更新时间：<span>{getFormatDate(post.updatedAt)}</span>
      </div>
      <div className="like">
        <span>
          <img alt="vote" src={like} />
        </span>
        <span>{post.vote}</span>
      </div>
    </li>
  );
}

export default PostItem;
```

7.2.5　帖子详情页

帖子详情页有两种状态，即浏览状态和编辑状态，分别对应图 7-2 和图 7-3。

在编辑状态下，对页面进行组件划分，分为 Header、Post、PostEditor、CommentList 和 CommentsView。其中，Header 和 PostEditor 已经实现，这里只分析 Post、CommentList 和 CommentsView。（为简化逻辑，帖子的点赞功能并未实现。）

1. Post

Post 负责获取帖子详情数据、修改帖子以及展示和创建帖子的评论。在组件挂载后，Post 调用后台 API 获取帖子详情和帖子的评论数据，代码如下：



图 7-2



图 7-3

```
componentDidMount() {
  this.refreshComments();
  this.refreshPost();
}
// 获取帖子详情
```

```
refreshPost() {
  const postId = this.props.match.params.id;
  get(url.getPostById(postId)).then(data => {
    if (!data.error && data.length === 1) {
      this.setState({
        post: data[0]
      });
    }
  });
}
 // 获取评论列表
refreshComments() {
  const postId = this.props.match.params.id;
  get(url.getCommentList(postId)).then(data => {
    if (!data.error) {
      this.setState({
        comments: data
      });
    }
  });
}
```

当 PostEditor 对帖子做了修改时，Post 会通过 handlePostSave 这个方法将更新的帖子同步到服务器。handlePostSave 代码如下：

```
handlePostSave(data) {
  const id = this.props.match.params.id;
  this.savePost(id, data);
}
// 同步帖子的修改到服务器
savePost(id, post) {
  put(url.updatePost(id), post).then(data => {
    if (!data.error) {
      /* 因为返回的帖子对象只有 author 的 id 信息，
       * 所有需要额外把完整的 author 信息合并到帖子对象中 */
      const newPost = { ...data, author: this.state.post.author };
      this.setState({
        post: newPost,
        editing: false
      });
    }
  });
}
```

当 CommentList 中有新的评论被创建时，Post 同样需要把新评论同步到服务器，这一过程通过 handleCommentSubmit 方法实现：

```
handleCommentSubmit(content) {
  const postId = this.props.match.params.id;
  const comment = {
    author: this.props.userId,
    post: postId,
    content: content
  };
  this.saveComment(comment);
}
// 保存新的评论到服务器
saveComment(comment) {
  post(url.createComment(), comment).then(data => {
    if (!data.error) {
      this.refreshComments();
    }
  });
}
```

components/Post.js 的完整代码如下：

```
import React, { Component } from "react";
import PostEditor from "./PostEditor";
import PostView from "./PostView";
import CommentList from "./CommentList";
import { get, put, post } from "../utils/request";
import url from "../utils/url";
import "./Post.css";

class Post extends Component {
  constructor(props) {
    super(props);
    this.state = {
      post: null,
      comments: [],
      editing: false
    };
    this.handleEditClick = this.handleEditClick.bind(this);
    this.handleCommentSubmit = this.handleCommentSubmit.bind(this);
    this.handlePostSave = this.handlePostSave.bind(this);
    this.handlePostCancel = this.handlePostCancel.bind(this);
    this.refreshComments = this.refreshComments.bind(this);
```

```
    this.refreshPost = this.refreshPost.bind(this);
  }

  componentDidMount() {
    this.refreshComments();
    this.refreshPost();
  }

  // 获取帖子详情
  refreshPost() {
    const postId = this.props.match.params.id;
    get(url.getPostById(postId)).then(data => {
      if (!data.error && data.length === 1) {
        this.setState({
          post: data[0]
        });
      }
    });
  }

  // 获取评论列表
  refreshComments() {
    const postId = this.props.match.params.id;
    get(url.getCommentList(postId)).then(data => {
      if (!data.error) {
        this.setState({
          comments: data
        });
      }
    });
  }

  // 让帖子处于编辑态
  handleEditClick() {
    this.setState({
      editing: true
    });
  }

  // 保存帖子
  handlePostSave(data) {
    const id = this.props.match.params.id;
    this.savePost(id, data);
  }

  // 取消编辑帖子
```

```
handlePostCancel() {
  this.setState({
    editing: false
  });
}

// 提交新建的评论
handleCommentSubmit(content) {
  const postId = this.props.match.params.id;
  const comment = {
    author: this.props.userId,
    post: postId,
    content: content
  };
  this.saveComment(comment);
}

// 保存新的评论到服务器
saveComment(comment) {
  post(url.createComment(), comment).then(data => {
    if (!data.error) {
      this.refreshComments();
    }
  });
}

// 同步帖子的修改到服务器
savePost(id, post) {
  put(url.updatePost(id), post).then(data => {
    if (!data.error) {
      /* 因为返回的帖子对象只有 author 的 id 信息,
       * 所有需要额外把完整的 author 信息合并到帖子对象中 */
      const newPost = { ...data, author: this.state.post.author };
      this.setState({
        post: newPost,
        editing: false
      });
    }
  });
}

render() {
  const { post, comments, editing } = this.state;
  const { userId } = this.props;
  if (!post) {
```

```
      return null;
    }
    const editable = userId === post.author.id;
    return (
      <div className="post">
        {editing ? (
          <PostEditor
            post={post}
            onSave={this.handlePostSave}
            onCancel={this.handlePostCancel}
          />
        ) : (
          {/* PostView 负责展示某一个帖子*/}
          <PostView
            post={post}
            editable={editable}
            onEditClick={this.handleEditClick}
          />
        )}
        <CommentList
          comments={comments}
          editable={Boolean(userId)}
          onSubmit={this.handleCommentSubmit}
        />
      </div>
    );
  }
}

export default Post;
```

2. CommentList

CommentList 用于显示评论列表（通过 CommentsView）和发表新评论，用户发表的新评论通过调用父组件 Post 的 handleCommentSubmit 方法保存到服务器。CommentList 也是只负责 UI 逻辑。components/CommentList.js 的完整代码如下：

```
import React, { Component } from "react";
import CommentsView from "./CommentsView";
import { getFormatDate } from "../utils/date";
import "./CommentList.css";

class CommentList extends Component {
  constructor(props) {
    super(props);
```

```
    this.state = {
      value: ""
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleClick = this.handleClick.bind(this);
  }

  // 处理新评论内容的变化
  handleChange(e) {
    this.setState({
      value: e.target.value
    });
  }

  // 保存新评论
  handleClick(e) {
    const content = this.state.value;
    if (content.length > 0) {
      this.props.onSubmit(this.state.value);
      this.setState({
        value: ""
      });
    } else {
      alert("评论内容不能为空！");
    }
  }

  render() {
    const { comments, editable } = this.props;

    return (
      <div className="commentList">
        <div className="title">评论</div>
        {/* 只有登录状态，才允许新建评论 */}
        {editable ? (
          <div className="editor">
            <textarea
              placeholder="说说你的看法"
              value={this.state.value}
              onChange={this.handleChange}
            />
            <button onClick={this.handleClick}>提交</button>
          </div>
        ) : null}
        <CommentsView comments={comments} />
```

```
      </div>
    );
  }
}

export default CommentList;
```

3. CommentsView

CommentsView 是负责显示评论列表的最终组件。components/CommentsView.js 的代码如下：

```
import React, { Component } from "react";
import { getFormatDate } from "../utils/date";
import "./CommentsView.css";

class CommentsView extends Component {
  render() {
    const { comments } = this.props;
    return (
      <ul className="commentsView">
        {comments.map(item => {
          return (
            <li key={item.id}>
              <div>{item.content}</div>
              <div className="sub">
                <span>{item.author.username}</span>
                <span>·</span>
                <span>{getFormatDate(item.updatedAt)}</span>
              </div>
            </li>
          );
        })}
      </ul>
    );
  }
}

export default CommentsView;
```

7.3　代码分片

默认情况下，当在项目根路径下执行 npm run build 时，create-react-app 内部使用 webpack 将 src/路径下的所有代码打包成一个 JS 文件和一个 CSS 文件。命令执行完成后，控制台有类似如下输出信息（两个文件名中的哈希值部分可能会与这里的输出有所不同）：

```
$ react-scripts build
Creating an optimized production build...
Compiled successfully.

File sizes after gzip:

  63.09 KB  build/static/js/main.8fdd292e.js
  699 B     build/static/css/main.58ed99f6.css
```

当项目代码量不多时，把所有代码打包到一个文件的做法并不会有什么影响。但是，对于一个大型应用，如果还把所有的代码都打包到一个文件中，显然就不合适了。试想，当用户访问登录页面时，浏览器加载的 JS 文件还包含其他页面的代码，这会延长网页的加载时间，给用户带来不好的体验。理想情况下，当用户访问一个页面时，该页面应该只加载自己使用到的代码。解决这个问题的方案就是代码分片，将 JS 代码分片打包到多个文件中，然后在访问页面时按需加载。

create-react-app 支持通过动态 import()的方式实现代码分片。import()接收一个模块的路径作为参数，然后返回一个 Promise 对象，Promise 对象的值就是待导入的模块对象。例如：

```
// moduleA.js
const moduleA = 'Hello';
export { moduleA };

//App.js
import React, { Component } from 'react';

class App extends Component {

  handleClick = () => {
    // 使用 import 动态导入 moduleA.js
    import('./moduleA')
      .then(({ moduleA }) => {
        // 使用 moduleA
      })
      .catch(err => {
        // 处理错误
      });
  };

  render() {
    return (
      <div>
        <button onClick={this.handleClick}>加载 moduleA</button>
      </div>
    );
  }
}

export default App;
```

上面的代码会将 moduleA.js 和它所依赖的其他模块单独打包到一个 chunk 文件中，只有当用户点击了加载按钮，才开始加载这个 chunk 文件。

当项目中使用 React Router 时，一般会根据路由信息将项目代码分片，每个路由依赖的代码单独打包成一个 chunk 文件。我们创建一个函数统一处理这个逻辑：

```
import React, { Component } from "react";

// importComponent 是使用了 import()的函数
export default function asyncComponent(importComponent) {
  class AsyncComponent extends Component {
    constructor(props) {
      super(props);
      this.state = {
        component: null  //动态加载的组件
      };
    }

    componentDidMount() {
      importComponent().then((mod) => {
        this.setState({
          // 同时兼容 ES6 和 CommonJS 的模块
          component: mod.default ? mod.default : mod
        });
      });
    }

    render() {
      // 渲染动态加载的组件
      const C = this.state.component;
      return C ? <C {...this.props} /> : null;
    }
  }

  return AsyncComponent;
}
```

asyncComponent 接收一个函数参数 importComponent，importComponent 内通过 import()语法动态导入模块。在 AsyncComponent 被挂载后，importComponent 就会被调用，进而触发动态导入模块的动作。

下面我们利用 asyncComponent 改造 BBS 项目，使其支持按照路由进行代码分片。本节项目源代码的目录为/chapter-07/bbs-router-code-splitting。在 App.js 中，删除使用 import 静态导入的 Login 和 Home 组件，改为使用 asyncComponent 动态导入，代码如下：

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch } from "react-router-dom";
```

```
import asyncComponent from "./AsyncComponent";
//通过 asyncComponent 导入组件，创建代码分片点
const AsyncHome = asyncComponent(() => import("./components/Home"));
const AsyncLogin = asyncComponent(() => import("./components/Login"));

class App extends Component {
  render() {
    return (
      <Router>
        <Switch>
          <Route exact path="/" component={AsyncHome} />
          <Route path="/login" component={AsyncLogin} />
          <Route path="/posts" component={AsyncHome} />
        </Switch>
      </Router>
    );
  }
}

export default App;
```

这样，只有当路由匹配时，对应的组件才会被导入，实现按需加载的效果。同样，Home.js 中使用的 Route 也进行相同改造，关键代码如下：

```
//Home.js

const AsyncPost = asyncComponent(() => import("./Post"));
const AsyncPostList = asyncComponent(() => import("./PostList"));

class Home extends Component {
  /** 省略其余代码 **/

  render() {
    const { match, location } = this.props;
    const { username } = this.state;
    return (
      <div>
        <Header
          username={username}
          onLogout={this.handleLogout}
          location={location}
        />
        <Route
          path={match.url}
          exact
```

```
          render={props => <AsyncPostList username={username} {...props} />}
        />
        <Route
          path={`${match.url}/:id`}
          render={props => <AsyncPost username={username} {...props} />}
        />
      </div>
    );
  }
}
```

看到这里，有些读者可能会有这样的疑问，为什么 asyncComponent 不直接接收一个代表组件路径的字符串作为参数，然后在 AsyncComponent 组件内部使用 import()动态导入该组件呢？这种情况下，实现代码如下：

```
export default function asyncComponent(componentPath) {
  class AsyncComponent extends Component {
    /** 省略其余代码 **/

    componentDidMount() {
      import(componentPath).then((mod) => {
        this.setState({
          // 同时兼容 ES6 和 CommonJS 的模块
          component: mod.default ? mod.default : mod
        });
      });
    }
  }

  return AsyncComponent;
}

// 使用 asyncComponent
const AsyncHome = asyncComponent("./components/Home");
```

如上修改后，重新编译打包，代码并没有被成功分片，控制台上还会有这样一句警告信息：Critical dependency: the request of a dependency is an expression。这是因为在使用 import()时，必须显式地声明要导入的组件路径，webpack 在打包时，会根据这些显式的声明拆分代码，否则，webpack 无法获得足够的关于拆分代码的信息。

现在对改造后的项目再次执行 npm run build，将会生成 1 个 main.js 文件和 4 个 chunk.js 文件。每一个 import()语法都会打包出一个 chunk 文件，项目中共使用了 4 次 import()，因此最终有 4 个 chunk.js 文件。控制台有类似如下输出信息：

```
$ react-scripts build
Creating an optimized production build...
```

```
Compiled successfully.

File sizes after gzip:

  59.81 KB  build/static/js/main.515e12e5.js
  4.41 KB   build/static/js/0.f4df54c2.chunk.js
  3.82 KB   build/static/js/3.0420c765.chunk.js
  3.56 KB   build/static/js/1.67500497.chunk.js
  3.28 KB   build/static/js/2.7eabe0af.chunk.js
```

这里还有一个需要注意的地方，打包后没有单独的 CSS 文件了。这是因为 CSS 样式被打包到各个 chunk 文件中，当 chunk 文件被加载执行时，会动态地把 CSS 样式插入页面中。如果希望把 chunk 中的 CSS 打包到一个单独的文件中，就需要修改 webpack 使用的 ExtractTextPlugin 插件的配置，但 create-react-app 并没有直接把 webpack 的配置文件暴露给用户，为了修改相应配置，需要将 create-react-app 管理的配置文件“弹射”出来，在项目根路径下执行：

```
npm run eject
```

项目中会多出两个文件夹：config 和 scripts，scripts 中包含项目启动、编译和测试的脚本，config 中包含项目使用的配置文件，webpack 的配置文件就在这个路径下，如图 7-4 所示。



图 7-4

打包 webpack.config.prod.js，找到配置 ExtractTextPlugin 的地方，添加 allChunks: true 这项配置：

```
new ExtractTextPlugin({
    filename: cssFilename,
    allChunks: true,          // 新加配置项
}),
```

然后重新编译项目，各个 chunk 文件使用的 CSS 样式又会统一打包到 main.css 中。


注　意

npm run eject 是一个不可逆操作，一旦将配置“弹射”出，就不能再回到之前的状态，配置的维护和修改工作将全权交给用户。不过，create-react-app 官方已经计划在 2.0 版本中添加 allChunks: true 这一配置项，届时将不再需要通过“弹射”的方式添加这个配置。

7.4　本章小结

本章先介绍了单页面应用和前端路由的概念，由此延伸到 React Router 这个 React 技术栈中最常用的前端路由解决方案。React Router 4 遵循 React 一切皆组件的思想，支持在任意组件中定义路由 Route 组件，更加灵活的使用方式也增加了使用难度。然后，本章通过 BBS 项目展示了 React Router 在实际项目中的使用方式。本章的最后还介绍了如何进行代码分片，代码分片的目的是实现代码的按需加载，提高应用的加载速度。

第 8 章

Redux：可预测的状态管理机

React 主要的关注点是如何创建可复用的视图层组件，对于组件之间的数据传递和状态管理并没有给出很好的解决方案，所以，当我们创建大型 Web 应用时，只使用 React 往往不能高效、清晰地对应用和组件状态进行管理，这时候就需要引入额外的类库完成这项工作，Redux 就是其中的一个代表。Redux 的思想继承自 Facebook 官方的 Flux 架构，但比 Flux 更加简洁易用。本章就来介绍一下 Redux 的原理和使用。

8.1 简　介

8.1.1 基本概念

随着单页面应用需求越来越复杂，应用需要管理的状态也变得越来越多。这里的状态不仅包括从服务器端获取的数据，还包括本地创建的数据，同样也包括反映 UI 状态的数据，例如当前路由的位置、选中的标签、弹出框的控制等。Redux 通过一系列约定的规范将修改应用状态的步骤标准化，让应用状态的管理不再错综复杂，而是如同一根长线般清晰。

下面通过一个简单的例子阐述 Redux 的核心概念。本章项目源代码的目录为/chapter-08/todos-redux。假设我们要开发一个待办事项 todos 的应用，用一个 JavaScript 对象描述这个应用的状态：

```
{
  todos: [{
    text: 'Learn React',
```

```
    completed: true
  }, {
    text: 'Learn Redux',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

todos 包含所有的事项，包括已经完成的和尚未完成的，根据 completed 属性区分该事项是否已经完成；visibilityFilter 是一个过滤条件，表示当前界面上应该显示哪些事项。这个对象需要被视为只读对象，当需要修改应用状态时，必须发送一个 action，action 描述应用状态应该如何修改，其实 action 只是一个普通的 JavaScript 对象。例如，当新增一个待办事项时，可以发送下面的 action：

```
{ type: 'ADD_TODO', text: 'Learn MobX' }
```

type 代表 action 的类型，此处 ADD_TODO 表示要新增一个待办事项，text 包含新增事项的内容。类似地，如果要让界面显示所有的事项，可以发送下面的 action:

```
{ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }
```

注意，action 的结构并不是确定的（但必须包含 type 字段），不同的人有不同的描述习惯，例如，可以通过 action 描述新增一个待办事项：

```
{ type: 'ADD', data: 'Learn MobX' }
```

只要保证你的程序知道如何解析定义的 action 即可。那么，如何解析 action 呢？Redux 通过 reducer 解析 action。reducer 是一个普通的 JavaScript 函数，接收 action 为参数，然后返回一个新的应用状态 state。例如，这里我们定义一个 reducer（省略处理 state 的逻辑代码）：

```
function todoApp(state = {}, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      // return new state
    case ADD_TODO:
      // return new state
    case TOGGLE_TODO:
      // return new state
    default:
      return state
  }
}
```

todoApp 就是管理应用全局 state 的 reducer，todoApp 每次返回的 state 的结构就是最初定义的应用状态的结构：

```
{
  todos: [...],
```

```
  visibilityFilter: ...
}
```

这一系列过程描述了 Redux 的基本概念。Redux 的主要思想就是描述应用的状态如何根据 action 进行更新，Redux 通过提供一系列 API 将这一主要思想的落地实施进行标准化和规范化。但就 Redux 的使用者而言，编写的绝大部分代码都是普通的 JavaScript 代码，只有在个别地方会用到 Redux 的 API 而已。

8.1.2　三大原则

Redux 应用需要遵循三大原则，否则程序很容易出现难以察觉的问题。

1. 唯一数据源

Redux 应用只维护一个全局的状态对象，存储在 Redux 的 store 中。唯一数据源是一种集中式管理应用状态的方式，便于监控任意时刻应用的状态和调试应用，减少出错的可能性。

2. 保持应用状态只读

在任何时候都不能直接修改应用状态。当需要修改应用状态时，必须发送一个 action，由这个 action 描述如何修改应用状态。这一看似烦琐的修改状态的方式实际上是 Redux 状态管理流程的核心，保证了在大型复杂应用中状态管理的有序进行。

3. 应用状态的改变通过纯函数完成

action 表明修改应用状态的意图，真正对应用状态做修改的是 reducer。reducer 必须是纯函数，所以 reducer 在接收到 action 时，不能直接修改原来的状态对象，而是要创建一个新的状态对象返回。


注　意

纯函数是指满足以下两个条件的函数：

（1）对于同样的参数值，函数的返回结果总是相同的，即该函数结果不依赖任何在程序执行过程中可能改变的变量。

（2）函数的执行不会产生副作用，例如修改外部对象或输出到 I/O 设备。

8.2　主要组成

通过前面的介绍可以发现 Redux 应用的主要组成有 action、reducer 和 store。下面借助 todos 这个示例详细地绍这三部分。

8.2.1　action

action 是 Redux 中信息的载体，是 store 唯一的信息来源。把 action 发送给 store 必须通过 store 的 dispatch 方法。action 是普通的 JavaScript 对象，但每个 action 必须有一个 type 属性描述 action 的类型，type 一般被定义为字符串常量。除了 type 属性外，action 的结构完全由自己决定，但应该

确保 action 的结构能清晰地描述实际业务场景。一般通过 action creator 创建 action，action creator 是返回 action 的函数。例如，下面是一个新增待办事项的 action creator：

```
function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}
```

todos 应用涉及的操作有新增待办事项、修改待办事项的状态（已完成/未完成）、筛选当前显示的待办事项列表。对应的完整 action creator 如下：

```
// actions.js

// action types
export const ADD_TODO = 'ADD_TODO'
export const TOGGLE_TODO = 'TOGGLE_TODO'
export const SET_VISIBILITY_FILTER = 'SET_VISIBILITY_FILTER'

// 筛选待办事项列表的条件
export const VisibilityFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
}

// action creators
// 新增待办事项
export function addTodo(text) {
  return { type: ADD_TODO, text }
}

// 修改某个待办事项的状态，index 是待办事项在 todos 数组中的位置索引
export function toggleTodo(index) {
  return { type: TOGGLE_TODO, index }
}

// 筛选当前显示的待办事项列表
export function setVisibilityFilter(filter) {
  return { type: SET_VISIBILITY_FILTER, filter }
}
```

8.2.2 reducer

action 用于描述应用发生了什么操作，reducer 则根据 action 做出响应，决定如何修改应用的状态 state。既然是修改 state，那么就应该在编写 reducer 前设计好 state。state 既可以包含服务器端获

取的数据，也可以包含 UI 状态，前面已经设计了 todos 应用的 state：

```
{
  todos: [{
    text: 'Learn React',
    completed: true
  }, {
    text: 'Learn Redux',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

有了 state，我们再为 state 编写 reducer。reducer 是一个纯函数，它接收两个参数，当前的 state 和 action，返回新的 state。reducer 函数签名如下：

```
(previousState, action) => newState
```

我们先来创建一个最基本的 reducer：

```
import { VisibilityFilters } from './actions'

const initialState = {
  todos: [],
  visibilityFilter: VisibilityFilters.SHOW_ALL
}

// reducer
function todoApp(state = initialState, action) {
  return state
}
```

todoApp 这个 reducer 不做任何事情，对于任意 action 做出的响应都是直接返回前一个 state。这里需要注意 state 初始值的设置，当 todoApp 第一次被调用时，state 等于 undefined，这时会用 initialState 初始化 state。现在为 todoApp 添加处理 type 等于 SET_VISIBILITY_FILTER 的 action，要做的事情是改变 state 的 visibilityFilter：

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return { ...state, visibilityFilter: action.filter }
    default:
      return state
  }
}
```

注意，这里使用 ES6 的扩展运算符（...）创建新的 state 对象，避免直接修改之前的 state 对象。还有一种常见的写法是使用 ES6 的 Object.assign()函数：

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

下面再来处理另外两个 action，同样需要保证每次返回的 state 对象都是一个新的对象：

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return { ...state, visibilityFilter: action.filter }
    // 新增待办事项
    case ADD_TODO:
      // 使用了 ES6 的扩展语法
      return { ...state,
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      }
    // 修改待办事项的状态（已完成/未完成）
    case TOGGLE_TODO:
      return { ...state,
        todos: state.todos.map((todo, index) => {
          if (index === action.index) {
            return { ...todo, completed: !todo.completed }
          }
          return todo
        })
      }
    default:
      return state
  }
}
```

当前，我们使用 todoApp 一个 reducer 处理所有的 action，当应用变得复杂时，这个 reducer 也会逐渐变复杂，这时，一般会拆分出多个 reducer，每个 reducer 处理 state 中的部分状态。例如，这里可以拆分出 todos 和 visibilityFilter 两个 reducer，分别处理 state 的 todos 和 visibilityFilter 两个子状态：

```
// 处理 todos 的 reducer
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([{ text: action.text, completed: false }])
    case 'TOGGLE_TODO':
      return state.map(
        (todo, index) =>
          action.index === index
            ? { ...todo, completed: !todo.completed }
            : todo
      )
    default:
      return state
  }
}

// 处理 visibilityFilter 的 reducer
function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
}

todoApp 简化为:
function todoApp(state = {}, action) {
  return {
    todos: todos(state.todos, action),
    visibilityFilter: visibilityFilter(state.visibilityFilter, action)
  }
}
```

注意，每个拆分的 reducer 只接收它负责的 state 中的部分属性，而不再是完整的 state 对象。todos 接收 state.todos，visibilityFilter 接收 state.visibilityFilter。这样，当应用较复杂时，就可以拆分出多个 reducer 保存到独立的文件中。

Redux 还提供了一个 combineReducers 函数，用于合并多个 reducer。使用 combineReducers，todoApp 可以改写如下：

```
import { combineReducers } from 'redux'

const todoApp = combineReducers({
  todos,
  visibilityFilter
})
```

它等价于：

```
function todoApp(state = {}, action) {
  return {
    todos: todos(state.todos, action),
    visibilityFilter: visibilityFilter(state.visibilityFilter, action)
  }
}
```

还可以为 combineReducers 接收的参数对象指定和 reducer 的函数名不同的 key 值：

```
const reducer = combineReducers({
  a: doSomethingWithA,
  b: processB,
  c: c
})
```

它等价于：

```
function reducer(state = {}, action) {
  return {
    a: doSomethingWithA(state.a, action),
    b: processB(state.b, action),
    c: c(state.c, action)
  }
}
```

可见，combineReducers 传递给每个 reducer 的 state 中的属性取决于它的参数对象的 key 值。

8.2.3 store

store 是 Redux 中的一个对象，也是 action 和 reducer 之间的桥梁。store 主要负责以下几个工作：

（1）保存应用状态。
（2）通过方法 getState()访问应用状态。
（3）通过方法 dispatch(action)发送更新状态的意图。
（4）通过方法 subscribe(listener)注册监听函数、监听应用状态的改变。

一个 Redux 应用中只有一个 store，store 保存了唯一数据源。store 通过 createStore()函数创建，创建时需要传递 reducer 作为参数，创建 todos 应用的 store 的代码如下：

```
import { createStore } from 'redux'
import todoApp from './reducers'

let store = createStore(todoApp)
```

创建 store 时还可以设置应用的初始状态：

```
// initialState 代表初始状态
let store = createStore(todoApp, initialState)
```

除了可以在创建 store 时设置应用的初始状态外，还可以在创建 reducer 时设置应用的初始状态，例如：

```
// 初始状态是一个空数组
function todos(state = [], action) {
 //...
}

// 初始状态等于 SHOW_ALL
function visibilityFilter(state = 'SHOW_ALL', action) {
 //...
}
```

todos 设置的初始状态是 state = []，visibilityFilter 设置的初始状态是 state = 'SHOW_ALL'，这样，当把这两个 reducer 合并成一个 reducer 时，两个 reducer 的初始状态就构成了整个应用的初始状态：

```
{
  todos: [],
  visibilityFilter: 'SHOW_ALL'
}
```

store 创建完成后，就可以通过 getState()获取当前应用的状态 state：

```
const state = store.getState()
```

当需要修改 state 时，通过 store 的 dispatch 方法发送 action。例如，发送一个新增待办事项的 action：

```
// 定义 action
function addTodo(text){
  return {type: 'ADD_TODO', text}
}

// 发送 action
store.dispatch(addTodo('Learn about actions'))
```

当 todoApp 这个 reducer 处理完成 addTodo 这个 action 时，应用的状态会被更新，此时通过 store.getState()可以得到最新的应用状态。为了能准确知道应用状态更新的时间，需要向 store 注册一个监听函数：

```
let unsubscribe = store.subscribe(() =>
  console.log(store.getState())
)
```

这样，每当应用状态更新时，最新的应用状态就会被打印出来。当需要取消监听时，直接调用 store.subscribe 返回的函数即可：

```
unsubscribe()
```

下面再来总结一下 Redux 的数据流过程。

（1）调用 store.dispatch(action)。一个 action 是一个用于描述“发生了什么”的对象。store.dispatch(action)可以在应用的任何地方调用，包括组件、XHR 的回调，甚至在定时器中。

（2)Redux 的 store 调用 reducer 函数。store 传递两个参数给 reducer：当前应用的状态和 action。reducer 必须是一个纯函数，它的唯一职责是计算下一个应用的状态。

（3）根 reducer 会把多个子 reducer 的返回结果组合成最终的应用状态。根 reducer 的构建形式完全取决于用户。Redux 提供了 combineReducers，方便把多个拆分的子 reducer 组合到一起，但完全可以不使用它。当使用 combineReducers 时，action 会传递给每一个子 reducer 处理，子 reducer 处理后的结果会合并成最终的应用状态。

（4）Redux 的 store 保存根 reducer 返回的完整应用状态。此时，应用状态才完成更新。如果 UI 需要根据应用状态进行更新，那么这就是更新 UI 的时机。对于 React 应用而言，可以在这个时候调用组件的 setState 方法，根据新的应用状态更新 UI。

8.3　在 React 中使用 Redux

8.3.1　安装 react-redux

首先需要强调，Redux 和 React 并无直接关联，Redux 可以和很多库一起使用。为了方便在 React 中使用 Redux，我们需要使用 react-redux 这个库。这个库并不包含在 Redux 中，所以需要单独安装：

```
npm install react-redux
```

8.3.2　展示组件和容器组件

根据组件意图的不同，可以将组件划分为两类：展示组件（presentational components）和容器组件（container components）。

展示组件负责应用的 UI 展示（how things look），也就是组件如何渲染，具有很强的内聚性。展示组件不关心渲染时使用的数据是如何获取到的，它只要知道有了这些数据后，组件应该如何渲染就足够了。数据如何获取是容器组件负责的事情。

容器组件负责应用逻辑的处理（how things work），如发送网络请求、处理返回数据、将处理过的数据传递给展示组件使用等。容器组件还提供修改源数据的方法，通过展示组件的 props 传递

给展示组件，当展示组件的状态变更引起源数据变化时，展示组件通过调用容器组件提供的方法同步这些变化。

展示组件和容器组件可以自由嵌套，一个容器组件可以包含多个展示组件和其他的容器组件；一个展示组件也可以包含容器组件和其他的展示组件。这样的分工可以使与 UI 渲染无直接关系的业务逻辑由容器组件集中负责，展示组件只关注 UI 的渲染逻辑，从而使展示组件更容易被复用。对于非常简单的页面，一般只需要一个容器组件就足够了；但对于复杂的页面，往往需要多个容器组件，否则所有的业务逻辑都在一个容器组件中处理的话，会导致这个组件非常复杂，同时这个组件获取到的源数据可能需要经过很多层组件 props 的传递才能到达最终使用的展示组件。


注 意

展示组件和容器组件容易和 2.2.4 小节介绍的无状态组件和有状态组件混淆。这两组概念对组件的划分依据是不同的。展示组件和容器组件是根据组件的意图划分组件，无状态组件和有状态组件是根据组件内部是否使用 state 划分组件。不过通常情况下，展示组件是通过无状态组件来实现的，容器组件是通过有状态组件来实现的，但是展示组件也可以是有状态组件，容器组件也可以是无状态组件。

8.3.3　connect

react-redux 提供了一个 connect 函数，用于把 React 组件和 Redux 的 store 连接起来，生成一个容器组件，负责数据管理和业务逻辑，代码如下：

```
import { connect } from 'react-redux'
import TodoList from './TodoList'

const VisibleTodoList = connect()(TodoList);
```

这里创建了一个容器组件 VisibleTodoList，可以把组件 TodoList 和 Redux 连接起来。但是，这个 VisibleTodoList 只是一个空壳，并没有负责任何真正的业务逻辑。根据 Redux 的数据流过程，VisibleTodoList 需要承担两个工作：

（1）从 Redux 的 store 中获取展示组件所需的应用状态。
（2）把展示组件的状态变化同步到 Redux 的 store 中。

通过为 connect 传递两个参数可以让 VisibleTodoList 具备这两个功能：

```
import { connect } from 'react-redux'
import TodoList from './TodoList'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)
```

mapStateToProps 和 mapDispatchToProps 的类型都是函数，前者负责从全局应用状态 state 中取出所需数据，映射到展示组件的 props，后者负责把需要用到的 action 映射到展示组件的 props 上。

8.3.4　mapStateToProps

mapStateToProps 是一个函数，从名字就可以看出，它的作用是把 state 转换成 props。state 就是 Redux store 中保存的应用状态，它会作为参数传递给 mapStateToProps，props 就是被连接的展示组件的 props。例如，VisibleTodoList 需要根据 state 中的 todos 和 visibilityFilter 两个数据过滤出传递给 TodoList 的待办事项数据：

```
function getVisibleTodos(todos, filter) {
  switch (filter) {
    case 'SHOW_ALL':
      return todos
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
  }
}

function mapStateToProps(state) {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}
```

每当 store 中的 state 更新时，mapStateToProps 就会重新执行，重新计算传递给展示组件的 props，从而触发组件的重新渲染。


注 意

store 中的 state 更新一定会导致 mapStateToProps 重新执行，但不一定会触发组件 render 方法的重新执行。如果 mapStateToProps 新返回的对象和之前的对象浅比较（shallow comparison）相等，组件的 shouldComponentUpdate 方法就会返回 false，组件的 render 方法也就不会被再次触发。这是 react-redux 库的一个重要优化。

connect 可以省略 mapStateToProps 参数，这样 state 的更新就不会引起组件的重新渲染。

mapStateToProps 除了接收 state 参数外，还可以使用第二个参数，代表容器组件的 props 对象，例如：

```
// ownProps 是组件的 props 对象
function mapStateToProps(state, ownProps) {
  //...
}
```

8.3.5　mapDispatchToProps

容器组件除了可以从 state 中读取数据外，还可以发送 action 更新 state，这就依赖于 connect

的第二个参数 mapDispatchToProps。mapDispatchToProps 接收 store.dispatch 方法作为参数，返回展示组件用来修改 state 的函数，例如：

```
// toggleTodo(id)返回一个 action
function toggleTodo(id) {
  return { type: 'TOGGLE_TODO', id }
}

function mapDispatchToProps(dispatch) {
  return {
    onTodoClick: function(id) {
      dispatch(toggleTodo(id))
    }
  }
}
```

这样，展示组件内就可以调用 this.props.onTodoClick(id)发送修改待办事项状态的 action 了。另外，与 mapStateToProps 相同，mapDispatchToProps 也支持第二个参数，代表容器组件的 props。

8.3.6　Provider 组件

通过 connect 函数创建出容器组件，但这个容器组件是如何获取到 Redux 的 store？react-redux 提供了一个 Provider 组件，Provider 的部分示意代码如下（为了方便理解，这里的代码和 react-redux 中的代码并不完全一致）：

```
class Provider extends Component {
  getChildContext() {
    return {
      store: this.props.store
    };
  }

  render() {
    return this.props.children;
  }
}

Provider.childContextTypes = {
  store: React.PropTypes.object
}
```

Provider 组件需要接收一个 store 属性，然后把 store 属性保存到 context（如果忘记 context 的用法，可参考 4.3 节组件通信）。Provider 组件正是通过 context 把 store 传递给子组件的，所以使用 Provider 组件时，一般把它作为根组件，这样内层的任意组件才可以从 context 中获取 store 对象，代码如下：

```
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import todoApp from './reducers'
import App from './components/App'

let store = createStore(todoApp);

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

8.4 中间件与异步操作

8.4.1 中间件

中间件（Middleware）的概念常用于 Web 服务器框架中，例如 Node.js 的 Web 框架 Express，代表处理请求的通用逻辑代码，一个请求在经历中间件的处理后，才能到达业务逻辑代码层。多个中间件可以串联起来使用，前一个中间件的输出是下一个中间件的输入，整个处理过程就如同“管道”一般，如图 8-1 所示。



图 8-1

Redux 的中间件概念与此类似，Redux 的 action 可类比 Web 框架收到的请求，reducer 可类比 Web 框架的业务逻辑层，因此，Redux 的中间件代表 action 在到达 reducer 前经过的处理程序。实际上，一个 Redux 中间件就是一个函数。Redux 中间件增强了 store 的功能，我们可以利用中间件为 action 添加一些通用功能，例如日志输出、异常捕获等。我们可以通过改造 store.dispatch 增加日志输出的功能：

```
let next = store.dispatch
store.dispatch = function dispatchAndLog(action) {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}
```

上面的代码重新定义了 store.dispatch，在发送 action 前后都添加了日志输出，这就是中间件的雏形，对 store.dispatch 方法进行了改造，在发出 action 和执行 reducer 这两步之间添加其他功能。

注意，实际的中间件实现方式要远比上面的示例复杂，本书不涉及这块内容。实际项目中，往往是直接使用别人写好的中间件。例如，上面介绍的日志输出功能就可以使用专门的日志中间件 redux-logger（https://github.com/evgenyrodionov/redux-logger）。为 store 添加中间件支持的代码如下：

```
import { applyMiddleware, createStore } from 'redux';
import logger from 'redux-logger';
import reducer from './reducers';

const store = createStore(
  reducer,
  applyMiddleware(logger)
);
```

上面的代码先从 redux-logger 中引入日志中间件 logger，然后将它放入 applyMiddleware 方法中并传给 createStore，完成 store.dispatch 功能的加强。下面来看一下 applyMiddleware 这个函数做了些什么，代码如下：

```
import compose from './compose'

export default function applyMiddleware(...middlewares) {
  return (createStore) => (...args) => {
    const store = createStore(...args)
    let dispatch = store.dispatch
    let chain = []

    const middlewareAPI = {
      getState: store.getState,
      dispatch: (...args) => dispatch(...args)
    }
    chain = middlewares.map(middleware => middleware(middlewareAPI))
    dispatch = compose(...chain)(store.dispatch)

    return {
      ...store,
      dispatch
    }
  }
}
```

要想完全理解这段源码并不容易，建议读者先把握主线逻辑：applyMiddleware 把接收到的中间件放入数组 chain 中，然后通过 compose(...chain)(store.dispatch)定义加强版的 dispatch 方法，compose 是一个工具函数，compose(f, g, h)等价于(...args)=> f(g(h(args)))。另外需要注意，每一个中间件都接收一个包含 getState 和 dispatch 的参数对象，在利用中间件执行异步操作时，将会使用到这两个方法。

8.4.2 异步操作

异步操作在 Web 应用中是不可缺少的，其中最常见的异步操作是向服务器请求数据。目前，我们介绍的 Redux 的工作流是：发送 action，reducer 立即处理收到的 action，reducer 返回新的 state。这个流程并不涉及异步操作，Redux 中处理异步操作必须借助中间件的帮助。

redux-thunk（https://github.com/gaearon/redux-thunk）是处理异步操作最常用的中间件。使用 redux-thunk 的代码如下：

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import reducer from './reducers';

const store = createStore(
  reducer,
  applyMiddleware(thunk)
);
```

现在定义一个异步 action 模拟向服务器请求数据：

```
// 异步 action
function getData(url){
  return function (dispatch) {
    return fetch(url)
      .then(
        response => response.json(),
        error => console.log('An error occured.', error)
      )
      .then(json =>
        dispatch({type:'RECEIVE_DATA', data: json})
      )
  }
}
```

发送这个 action:

```
store.dispatch(getData("http://xxx"));
```

不使用 redux-thunk 中间件时，上面的代码会报错，因为 store.dispatch 只能接收普通 JavaScript 对象代表的 action，现在使用 redux-thunk，store.dispatch 就能接收函数作为参数了。异步 action 会先经过 redux-thunk 的处理，当请求返回后，再次发送一个 action：dispatch({type:'RECEIVE_DATA', json})，把返回的数据发送出去，这时的 action 就是一个普通的 JavaScript 对象了，处理流程也和不使用中间件的流程一样。

在实际项目中，处理一个网络请求往往会使用三个 action，分别表示请求开始、请求成功和请求失败，例如：

```
{ type: 'FETCH_DATA_REQUEST' }
{ type: 'FETCH_DATA_SUCCESS', data: { ... } }
{ type: 'FETCH_DATA_FAILURE', error: 'Oops' }
```

使用这三个 action 改写上面的代码：

```
// 异步 action
function getData(url){
  return function (dispatch) {
    dispatch({type:'FETCH_DATA_REQUEST'});
    return fetch(url)
      .then(
        response => response.json(),
        error => {
          console.log('An error occured.', error);
          dispatch({type:'FETCH_DATA_FAILURE', error});
        }
      )
      .then(json =>
        dispatch({type:'FETCH_DATA_SUCCESS', data: json});
      )
  }
}
```

这样，应用就可以根据请求所处的阶段显示不同的 UI，例如控制 Loading 效果。

除了 redux-thunk 外，常用于处理异步操作的中间件还有 redux-promise（https://github.com/acdlite redux-promise）、redux-saga（https://github.com/redux-saga/redux-saga）等，本书不再展开介绍。

8.5　本章小结

本章详细介绍了 Redux 架构以及 Redux 各组成部分（action、reducer、store）的使用。在 React 项目中使用 Redux 需要借助 react-redux，它可以方便地将 React 组件和 Redux 的 store 连接。中间件是 Redux 的一大利器，Redux 中执行异步操作就是通过引入中间件实现的。

下一章，我们将在实战项目中使用 Redux。

第 9 章

Redux 项目实战

在学习了第 8 章 Redux 的基础知识后，当面对真实项目时，相信很多读者还是无从下手，不知道该如何使用 Redux。这是因为 Redux 本身的抽象程度很高，只关注最核心的状态管理功能，至于具体在项目中如何使用，Redux 更多的是把这个灵活度交给使用者，但这也给很多初学者带来困惑。本章将结合笔者自身的实践经验，从组织项目结构、设计应用 state 和设计 Redux 模块三方面介绍如何在真实项目中使用 Redux。但请注意，本章介绍的 Redux 并不是唯一的使用方式，只是笔者推荐的其中一种使用方式。本章最后介绍经常和 Redux 一起使用的另外两个库：Immutable.js 和 Reselect，它们可以提高 Redux 的性能。

本章依然使用 BBS 项目作为示例。本章项目的源码目录为：/chapter-09/bbs-redux。

9.1 组织项目结构

关于如何组织 React+Redux 的项目结构一直有多种声音，目前主流的方案有三种：按照类型、按照页面功能和 Ducks。

1. 按照类型

这里的类型指的是一个文件在项目中充当的角色类型，即这个文件是一个 component（展示组件）还是一个 container（容器组件），或者是一个 reducer 等，充当 component、container、action、reducer 等不同角色的文件分别放在不同的文件夹下，这也是 Redux 官方网站示例所采用的项目结构。这种结构如下：

```
actions/
  action1.js
  action2.js
components/
  component1.js
  component2.js
  component3.js
constainers/
  constainer1.js
  constainer2.js
reducers/
  reducer1.js
  reducer2.js
index.js
```

使用这种结构组织项目，每当增加一个新功能时，需要在 containers 和 components 文件夹下增加这个功能需要的组件，还要在 actions 和 reducers 文件夹下分别添加 Redux 管理这个功能使用到的 action 和 reducer，如果 action type 放在另一个文件夹，还需要在这个文件夹下增加新的 action type。所以，开发一个功能时，需要频繁地切换路径以修改不同的文件。如果项目比较小，例如第 8 章中的 todos 项目采用这种结构问题还不大，但对于一个规模较大的项目来说，使用这种项目结构是非常不方便的。

2. 按照页面功能

一个页面功能对应一个文件夹，这个页面功能所用到的 container、component、action、reducer 等文件都存放在这个文件夹下，如下所示：

```
feature1/
  components/
  actions.js
  container.js
  index.js
  reducer.js
feature2/
  components/
  actions.js
  container.js
  index.js
  reducer.js
index.js
rootReducer.js
```

这种项目结构的好处显而易见，一个页面功能使用到的组件、状态和行为都在同一个文件夹下，方便开发，易于功能的扩展，Github 上很多脚手架也选择了这种目录结构，如 https://github.com/react-boilerplate/react-boilerplate。但使用这种结构依然无法解决开发一个功能时，

需要频繁在 reducer、action、action type 等不同文件间切换的问题。另外，Redux 将整个应用的状态放在一个 store 中来管理，不同的功能模块之间可以共享 store 中的部分状态（项目越复杂，这种场景就会越多），共享的状态应该放到哪一个页面功能文件夹下也是一个问题。这些问题归根结底是因为 Redux 中的状态管理逻辑并不是根据页面功能划分的，它是页面功能之上的一层抽象。

3. Ducks

Ducks 指的是一种新的 Redux 项目结构的提议，这份提议的地址是：https://github.com/erikras/ducks-modular-redux。它提倡将相关联的 reducer、action types 和 action creators 写到一个文件里。本质上是以应用的状态作为划分模块的依据，而不是以界面功能作为划分模块的依据。这样，管理相同状态的依赖都在同一个文件中，无论哪个容器组件需要使用这部分状态，只需要引入管理这个状态的模块文件即可。这样的一个文件（模块）代码如下：

```
// widget.js

// Actions
const LOAD   = 'widget/LOAD';
const CREATE = 'widget/CREATE';
const UPDATE = 'widget/UPDATE';
const REMOVE = 'widget/REMOVE';

const initialState = {
  widget: null,
  isLoading: false,
}

// Reducer
export default function reducer(state = initialState, action = {}) {
  switch (action.type) {
    LOAD:
      //...
    CREATE:
      //...
    UPDATE:
      //...
    REMOVE:
      //...
    default: return state;
  }
}

// Action Creators
export function loadWidget() {
  return { type: LOAD };
}
```

```
export function createWidget(widget) {
  return { type: CREATE, widget };
}

export function updateWidget(widget) {
  return { type: UPDATE, widget };
}

export function removeWidget(widget) {
  return { type: REMOVE, widget };
}
```

整体的目录结构如下：

```
components/  (通用组件)
containers/
  feature1/
    components/  (feature1 的专用组件)
    index.js     (feature1 的容器组件)
redux/
  index.js (combineReducers)
  module1.js (reducer, action types, action creators)
  module2.js (reducer, action types, action creators)
index.js
```

在前两种项目结构中，当 container 需要使用 actions 时，可以通过 import * as actions from 'path/to/actions.js'的方式一次性把一个 action 文件中的所有 action creators 都引入进来。但在使用 Ducks 结构时，action creators 和 reducer 定义在同一个文件中，import *的导入方式会把 reducer 也导入进来（如果 action types 也被 export，那么还会导入 action types）。为解决这个问题，可以把 action creators 和 action types 定义到一个命名空间中：

```
// widget.js

// Actions，定义到 types 命名空间下
export const types = {
  LOAD  : 'widget/LOAD',
  CREATE: 'widget/CREATE',
  UPDATE: 'widget/UPDATE',
  REMOVE: 'widget/REMOVE'
}

const initialState = {
  widget: null,
  isLoading: false,
}

// Reducer
```

```
export default function reducer(state = initialState, action = {}) {
  switch (action.type) {
    types.LOAD:
      //...
    types.CREATE:
      //...
    types.UPDATE:
      //...
    types.REMOVE:
      //...
    default: return state;
  }
}

// Action Creators，定义到 actions 命名空间下
export const actions = {
  loadWidget: function() {
    return { type: types.LOAD };
  },
  createWidget: createWidget(widget) {
    return { type: types.CREATE, widget };
  },
  updateWidget: function(widget) {
    return { type: types.UPDATE, widget };
  },
  removeWidget: function(widget) {
    return { type: types.REMOVE, widget };
  }
}
```

这样，在 container 中使用 action creators 时，可以通过 import { actions } from 'path/to/module.js' 引入，避免引入额外的对象，也避免逐个导入 action creator 的烦琐。

采用 Ducks 这种项目结构重新组织 BBS 项目的目录结构，最终的目录结构如图 9-1 所示。

这里，我们把 action types、action creators 和 reducer 组成的每一个模块都放到了 redux/modules 路径下，而不是直接放在 redux 文件夹下，一方面是因为 redux 文件夹下可能还需要放置其他与 redux 相关的模块，例如自定义的 Middleware；另一方面，增加一层 modules 文件夹更能体现其作为核心业务逻辑模块的意义。模块的划分方式及其具体内容接下来就会介绍。另外，components 文件夹下的很多页面专用组件移动到对应页面下的 components 文件夹中，例如，PostItem 移动到 PostList/components 下，PostView、PostEditor 和 CommentList 移动到 Post/components 下，如图 9-2 所示。src/components 中只保留具有全局通用性质的组件，例如页面加载效果组件 Loading、用于显示错误信息的模态框组件 ModalDialog 等。

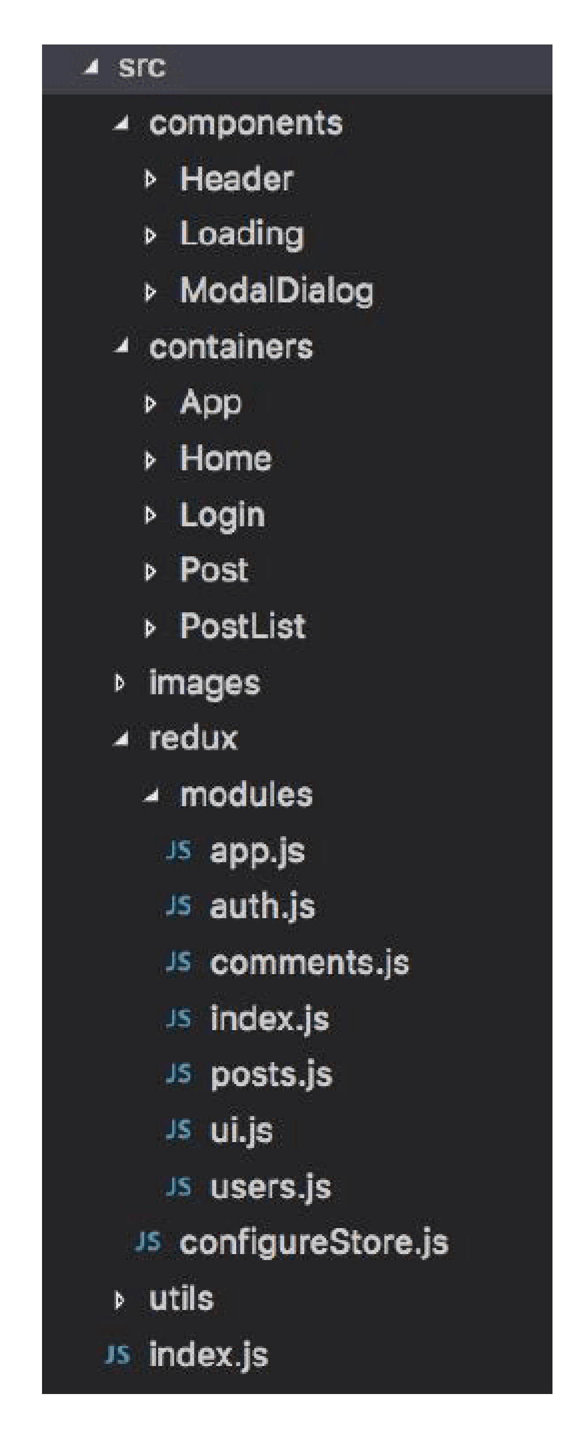


图 9-1

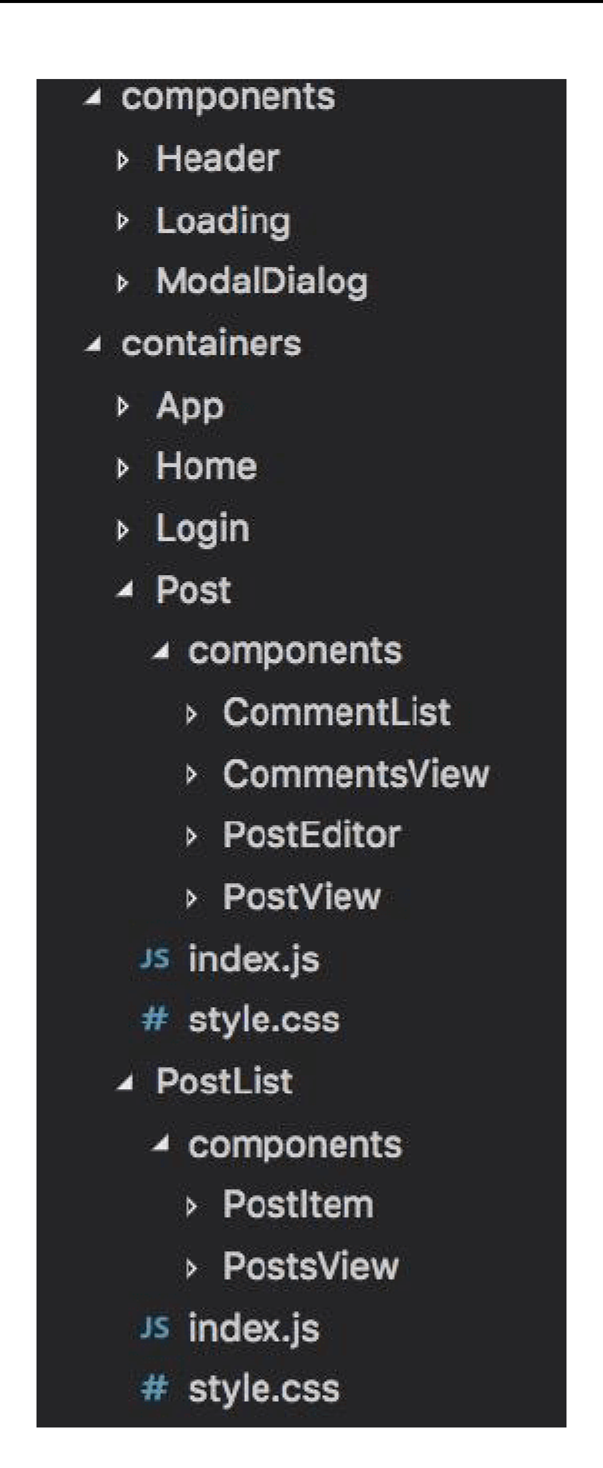


图 9-2

9.2　设计 state

Redux 应用执行过程中的任何一个时刻本质上都是该时刻的应用 state 的反映。可以说，state 驱动了 Redux 逻辑的运转。对于 Redux 项目来说，设计良好的 state 结构至关重要。下面先来看看设计 state 时容易犯的两个错误。

9.2.1　错误 1：以 API 作为设计 state 的依据

以 API 作为设计 state 的依据往往是一个 API 对应全局 state 中的一部分结构，且这部分结构同 API 返回的数据结构保持一致（或接近一致）。例如，在 BBS 项目中， 获取帖子列表 API 返回的数据结构如下：

```
[
  {
    id: "59f5cb68af52dd0b51be9503",
    title: "大家一起来讨论 React 吧",
    vote: 8,
    updatedAt: "2017-10-29T12:36:56.323Z",
    author: {
      id: "59e6f27aa9436dd037ea53ae",
      username: "steve"
```

```
      }
    },
    {
      id: "59f590696834b5f7614ed5ab",
      title: "前端框架，你最爱哪一个",
      vote: 2,
      updatedAt: "2017-10-26T08:48:56.856Z",
      author: {
        id: "59e5d22f6722f75272b3bbcf",
        username: "tom"
      }
    },
    {
      title: "Web App 的时代已经到来",
      vote: 10,
      updatedAt: "2017-10-26T08:48:56.856Z",
      author: {
        id: "59e5d22f6722f75272b3bbcf",
        username: "tom"
      }
    },
    ...
  ]
```

当查看帖子详情时，需要调用获取帖子详情 API 和获取帖子评论数据 API，两个接口返回的数据结构分别如下：

```
//帖子详情 API 返回数据结构
{
  id: "59f5cb68af52dd0b51be9503",
  title: "大家一起来讨论 React 吧",
  vote: 8,
  updatedAt: "2017-10-29T12:36:56.323Z",
  author: {
    id: "59e6f27aa9436dd037ea53ae",
    username: "steve"
  },
  content: "前端 UI 的复杂化，其本质问题是如何将来源于服务器端的动态数据和用户的交互行为高效地反映到复杂的用户界面上。React 另辟蹊径，通过引入虚拟 DOM、状态、单向数据流等设计理念，形成以组件为核心，用组件搭建 UI 的开发模式理顺了 UI 的开发过程，完美地将数据、组件状态和 UI 映射到一起，极大地提高了开发大型 Web 应用的效率。"
}
//评论列表 API 返回数据结构
[
```

```
  {
    id: "59f1a17ffb49aaf956b46215",
    content: "大爱 React！",
    updatedAt: "2017-10-26T08:49:03.235Z",
    author: {
      id: "59e5d22f6722f75272b3bbcf",
      username: "tom"
    }
  },
  {
    id: "59f1a17ffb49aaf956b46215",
    content: "React 大大提高了开发效率！",
    updatedAt: "2017-10-26T08:49:03.235Z",
    author: {
      id: "59e6f27aa9436dd037ea53ae",
      username: "steve"
    }
  },
  ...
]
```

上面三个接口的数据分别作为 state 的一部分，组合在一起构成应用全局的 state:

```
{
  posts: [
    {
      id: "59f5cb68af52dd0b51be9503",
      title: "大家一起来讨论 React 吧",
      vote: 8,
      updatedAt: "2017-10-29T12:36:56.323Z",
      author: {
        id: "59e6f27aa9436dd037ea53ae",
        username: "steve"
      }
    },
  ...
  ],
  currentPost: {
    id: "59f5cb68af52dd0b51be9503",
    title: "大家一起来讨论 React 吧",
    vote: 8,
    updatedAt: "2017-10-29T12:36:56.323Z",
    author: {
      id: "59e6f27aa9436dd037ea53ae",
```

```
        username: "steve"
      },
      content: "前端 UI 的复杂化，其本质问题是如何将来源于服务器端的动态数据和用户的交互行为高效地反映到复杂的用户界面上。React 另辟蹊径，通过引入虚拟 DOM、状态、单向数据流等设计理念，形成以组件为核心，用组件搭建 UI 的开发模式理顺了 UI 的开发过程，完美地将数据、组件状态和 UI 映射到一起，极大地提高了开发大型 Web 应用的效率。"
    },
    currentComments: [
      {
        id: "59f1a17ffb49aaf956b46215",
        content: "大爱 React！",
        updatedAt: "2017-10-26T08:49:03.235Z",
        author: {
          id: "59e5d22f6722f75272b3bbcf",
          username: "tom"
        }
      },
      ...
    ]
  }
```

这个 state 中，posts 和 currentPost 存在很多重复的信息，而且 posts、currentComments 是数组类型的结构，不便于查找，每次查找某条记录时，都需要遍历整个数组。这些问题本质上是因为 API 是基于服务端逻辑设计的，而不是基于应用的状态设计的。比如，虽然获取帖子列表时已经获取到帖子的标题、作者等基本信息，但对于获取帖子详情的 API 来说，根据 API 的设计原则，这个 API 依然应该包含这些基本信息，而不能只是返回帖子的正文内容。再比如，posts、currentComments 之所以返回数组结构，是考虑到数据的有序性、分页等因素。

9.2.2 错误 2：以页面 UI 为设计 state 的依据

既然不能依据 API 设计 state，很多人又会走另一条路，基于页面 UI 设计 state。页面 UI 需要什么样的数据和数据结构，state 就设计成什么样。以 todos 应用为例，页面会有三种状态：显示所有的事项、显然所有的已办事项和显示所有的待办事项。以页面 UI 为设计 state 的依据， state 将是这样的：

```
{
  all: [
    {
      id: 1,
      text: "todo 1",
      completed: false
    },
    {
      id: 2,
```

```
      text: "todo 2",
      completed: true
    }
  ],
  uncompleted: [
    {
      id: 1,
      text: "todo 1",
      completed: false
    }
  ],
  completed: [
    {
      id: 2,
      text: "todo 2",
      completed: false
    }
  ]
}
```

这个 state 对于展示 UI 的组件来说使用起来非常方便，当前应用处于哪种状态，就用对应状态的数组类型的数据渲染 UI，不用做任何中间数据转换。但这种 state 存在的问题也很容易被发现，一是这种 state 依然存在数据重复的问题；二是当新增或修改一条记录时，需要修改不止一个地方。例如，当新增一条记录时，all 和 uncompleted 这两个数组都要添加这条新增记录。这样设计的 state 既会造成存储的浪费，又会存在数据不一致的风险。

这两种设计 state 的方式实际上是两种极端，在实际项目中，完全按照这两种方式设计 state 的开发者并不多，但绝大部分人都会受到这两种设计方式的影响。

9.2.3　合理设计 state

看过了 state 的错误设计方式，下面来看一下应该如何合理地设计 state。设计 state 时，最重要的是记住一句话：像设计数据库一样设计 state。把 state 看作一个数据库，state 中的每一部分状态看作数据库中的一张表，状态中的每一个字段对应表的一个字段。设计一个数据库应该遵循以下三个原则：

（1）数据按照领域（Domain）分类存储在不同的表中，不同的表中存储的列数据不能重复。
（2）表中每一列的数据都依赖于这张表的主键。
（3）表中除了主键以外，其他列互相之间不能有直接依赖关系。

根据这三个原则可以翻译出设计 state 时的原则：

（1）把整个应用的状态按照领域分成若干子状态，子状态之间不能保存重复的数据。
（2）state 以键值对的结构存储数据，以记录的 key 或 ID 作为记录的索引，记录中的其他字段都依赖于索引。

（3）state 中不能保存可以通过 state 中的已有字段计算而来的数据，即 state 中的字段不互相依赖。

按照这三个原则重新设计 BBS 的 state。按领域划分，state 可以拆分为三个子 state：posts、comments 和 users，posts 中的记录以帖子的 id 为 key 值，结构如下：

```
posts: {
  59f5cb68af52dd0b51be9503: {
    id: "59f5cb68af52dd0b51be9503",
    title: "大家一起来讨论 React 吧",
    vote: 8,
    updatedAt: "2017-10-29T12:36:56.323Z",
    author: "59e6f27aa9436dd037ea53ae"
  },
  59f590696834b5f7614ed5ab: {
    id: "59f590696834b5f7614ed5ab",
    title: "前端框架，你最爱哪一个",
    vote: 2,
    updatedAt: "2017-10-29T12:36:56.323Z",
    author: "59e5d22f6722f75272b3bbcf"
  },
  ...
}
```

这个结构相比前面的按 API 划分 state 结构变化之处主要有两点：第一点是，posts 中的数据类型由数组类型改为以帖子 id 为 key 的 JSON 对象类型；第二点是，author 字段不再存储完整的作者信息，只存储作者的 id。第一个变化可以方便在使用 posts 时快速根据 id 获取对应帖子数据；第二个变化把原本嵌套的数据结构扁平化，避免了查询和修改嵌套数据时需要向下访问多个层级的烦琐，同时扁平化的数据结构更利于扩展。

但这个 state 还有不满足应用需求的地方：键值对的存储方式无法保证数据的有序性，但对于帖子列表，有序性显然是需要的。解决这个问题可以通过定义一个数组类型的属性 allIds 存储帖子的 id，同时将之前的键值对类型的数据存储在 byId 属性下：

```
posts: {
  byId: {
    59f5cb68af52dd0b51be9503: {
      id: "59f5cb68af52dd0b51be9503",
      title: "大家一起来讨论 React 吧",
      vote: 8,
      updatedAt: "2017-10-29T12:36:56.323Z",
      author: "59e6f27aa9436dd037ea53ae"
    },
    59f590696834b5f7614ed5ab: {
      id: "59f590696834b5f7614ed5ab",
```

```
        title: "前端框架，你最爱哪一个",
        vote: 2,
        updatedAt: "2017-10-29T12:36:56.323Z",
        author: "59e5d22f6722f75272b3bbcf"
      },
    ...
    },
    allIds: ["59f5cb68af52dd0b51be9503","59f590696834b5f7614ed5ab", ...]
  },
```

这样一来，allIds 负责维护数据的有序性，byId 负责根据 id 快速查询对应数据。这种设计 state 的方式是很常用的一种方式，请读者注意。

posts 不再保存完整的作者信息，那么作者信息的查询就有赖于领域 users 对应的子 state。应用中不关注作者的顺序，因此我们只需要使用以作者 id 为 key 的键值对存储数据即可：

```
users: {
  59e5d22f6722f75272b3bbcf: {
    id: "59e5d22f6722f75272b3bbcf",
    username: "tom"
  },
  59e6f27aa9436dd037ea53ae: {
    id: "59e6f27aa9436dd037ea53ae",
    username: "steve"
  },
  ...
}
```

评论数据是通过单独的 API 获取的，但评论数据是从属于某个帖子的，这个关系应该如何在 state 中体现呢？有两种方法：第一种是在 posts 对应的 state 中增加一个 comments 属性，存储该帖子对应的评论数据的 id；第二种是在 comments 对应的 state 中增加一个 byPost 属性，存储以帖子 id 作为 key，以这个帖子下的所有评论 id 作为值的对象。使用第二种方法，当调用 API 请求评论数据时，只需要修改 comments 对应的 state 即可，使用第一种方法还需要修改 posts 对应的 state，因此这里使用第二种方法：

```
comments: {
  byPost: {
    59f5cb68af52dd0b51be9503: ["59f1a17ffb49aaf956b46215",
"59f1a17ffb49aaf956b46215", ...],
    ...
  },
  byId: {
    59f1a17ffb49aaf956b46215: {
      id: "59f1a17ffb49aaf956b46215",
      content: "大爱 React！",
```

```
      updatedAt: "2017-10-26T08:49:03.235Z",
      author: "59e5d22f6722f75272b3bbcf"
    },
    59f1a17ffb49aaf956b46215: {
      id: "59f1a17ffb49aaf956b46215",
      content: "React 大大提高了开发效率！",
      updatedAt: "2017-10-26T08:49:03.235Z",
      author: 59e6f27aa9436dd037ea53ae"
    },
    ...
  }
}
```

byPost 保存帖子 id 到评论 id 的映射关系，byId 保存评论 id 到评论数据的映射关系。

由 posts、comments 和 users 三个领域组成的 state 结构如下：

```
{
  posts: {
    byId: {
      ...
    },
    allIds: [...]
  },
  comments: {
    byPost: {
      ...
    },
    byId: {
      ...
    }
  },
  users: {
    ...
  }
}
```

到目前为止，我们的 state 都是根据领域数据进行设计的，但实际上，应用的 state 不仅包含领域数据，还包含应用状态数据和 UI 状态数据。应用状态数据指反映应用行为的数据，例如，当前登录的状态、是否有 API 请求在进行等。UI 状态数据是代表 UI 当前如何显示的数据，例如对话框当前是否处于打开状态等。


注 意

有些开发者习惯把 UI 状态数据仍然保存在组件的 state 中，由组件自己管理，而不是交给 Redux 管理。这也是一种可选的做法，但将 UI 状态数据也交给 Redux 统一管理有利于应用 UI 状态的追溯。

在 BBS 项目中，我们将应用状态分为两部分，一部分专门记录登录认证相关的状态，保存到子 state auth 中，其余应用状态保存到子 state app 中。这两部分 state 结构如下：

```
app: {
  requestQuantity: 0,   // 当前应用中正在进行的 API 请求数
  error: null           // 应用全局错误信息
},
auth: {
  userId: null,
  username: null
}
```

app 中保存了当前进行中的 API 请求数量和应用的错误信息，auth 中保存了当前登录的用户 ID 和用户名。当需要管理的应用状态数据增多时，可以进一步将 app 拆分成多个子 state。类似地，我们将 UI 状态数据保存到子 state ui 中：

```
ui: {
  addDialogOpen: false,     // 用于新增帖子的对话框的显示状态
  editDialogOpen: false     // 用于编辑帖子的对话框的显示状态
}
```

这里涉及的 UI 状态数据比较少，只保存了新增帖子对话框和编辑帖子对话框的状态。

至此，由领域数据、应用状态数据、UI 状态数据组成的完整 state 结构如下：

```
{
  posts: {
    byId: {
      ...
    },
    allIds: [...]
  },
  comments: {
    byPost: {
      ...
    },
    byId: {
      ...
    }
  },
  users: {
    ...
  },
  app: {
    ...
  },
```

```
  auth: {
    ...
  },
  ui: {
    ...
  }
}
```

9.3 设计模块

在 9.1 节中已经介绍过，一个功能相关的 reducer、action types、action creators 将定义到一个文件中，作为一个 Redux 模块。根据 state 的结构，我们可以拆分出 app、auth、posts、comments、users、ui 和 index 七个模块。下面就来逐一介绍。

9.3.1 app 模块

app 模块负责标记 API 请求的开始和结束以及应用全局错误信息的设置，app.js 代码如下：

```
const initialState = {
  requestQuantity: 0,      // 当前应用中正在进行的 API 请求数
  error: null               // 应用全局错误信息
};

// action types
export const types = {
  START_REQUEST: "APP/START_REQUEST",      // 开始发送请求
  FINISH_REQUEST: "APP/FINISH_REQUEST",    // 请求结束
  SET_ERROR: "APP/SET_ERROR",              // 设置错误信息
  REMOVE_ERROR: "APP/REMOVE_ERROR"         // 删除错误信息
};

// action creators
export const actions = {
  startRequest: () => ({
    type: types.START_REQUEST
  }),
  finishRequest: () => ({
    type: types.FINISH_REQUEST
  }),
  setError: error => ({
    type: types.SET_ERROR,
    error
```

```
    }),
    removeError: () => ({
      type: types.REMOVE_ERROR
    })
  };

  // reducers
  const reducer = (state = initialState, action) => {
    switch (action.type) {
      case types.START_REQUEST:
        // 每接收一个 API 请求开始的 action，requestQuantity 加 1
        return { ...state, requestQuantity: state.requestQuantity + 1 };
      case types.FINISH_REQUEST:
        // 每接收一个 API 请求结束的 action，requestQuantity 减 1
        return { ...state, requestQuantity: state.requestQuantity - 1 };
      case types.SET_ERROR:
        return { ...state, error: action.error };
      case types.REMOVE_ERROR:
        return { ...state, error: null };
      default:
        return state;
    }
  };

  export default reducer;
```

这里需要注意，app 模块中的 action creators 会被其他模块调用。例如，其他模块用于请求 API 的异步 action 中，需要在发送请求的开始和结束时分别调用 startRequest 和 finishRequest；在 API 返回错误信息时，需要调用 setError 设置错误信息。这也说明，我们定义的模块并非只能被 UI 组件使用，各个模块之间也是可以互相调用的。

9.3.2　auth 模块

auth 模块负责应用的登录和注销。登录会调用服务器 API 做认证，这时就涉及异步 action，我们使用前面介绍的 redux-thunk 定义异步 action。注销逻辑仍然简化处理，只是清除客户端的登录用户信息。auth.js 的主要代码如下：

```
import { post } from "../../utils/request";
import url from "../../utils/url";
import { actions as appActions } from "./app";

const initialState = {
  userId: null,
  username: null
};
```

```
// action types
export const types = {
  LOGIN: "AUTH/LOGIN",    //登录
  LOGOUT: "AUTH/LOGOUT"   //注销
};

// action creators
export const actions = {
    // 异步 action，执行登录验证
  login: (username, password) => {
    return dispatch => {
      // 每个 API 请求开始前，发送 app 模块定义的 startRequest action
      dispatch(appActions.startRequest());
      const params = { username, password };
      return post(url.login(), params).then(data => {
        // 每个 API 请求结束后，发送 app 模块定义的 finishRequest action
        dispatch(appActions.finishRequest());
        // 请求返回成功，保存登录用户的信息，否则，设置全局错误信息
        if (!data.error) {
          dispatch(actions.setLoginInfo(data.userId, username));
        } else {
          dispatch(appActions.setError(data.error));
        }
      });
    };
  },
  logout: () => ({
    type: types.LOGOUT
  }),
  setLoginInfo: (userId, username) => ({
    type: types.LOGIN,
    userId: userId,
    username: username
  })
};

// reducers
const reducer = (state = initialState, action) => {
  switch (action.type) {
    case types.LOGIN:
      return { ...state, userId: action.userId, username: action.username };
    case types.LOGOUT:
      return { ...state, userId: null, username: null };
```

```
    default:
      return state;
  }
};

export default reducer;
```

9.3.3　posts 模块

posts 模块负责与帖子相关的状态管理，包括获取帖子列表、获取帖子详情、新建帖子和修改帖子。使用到的 action types 定义如下：

```
// action types
export const types = {
  CREATE_POST: "POSTS/CREATE_POST",           //新建帖子
  UPDATE_POST: "POSTS/UPDATE_POST",           //修改帖子
  FETCH_ALL_POSTS: "POSTS/FETCH_ALL_POSTS",  //获取帖子列表
  FETCH_POST: "POSTS/FETCH_POST"              //获取帖子详情
};
```

相应地，我们需要定义如下 action creators：

```
// action creators
export const actions = {
  // 获取帖子列表
  fetchAllPosts: () => {
    return (dispatch, getState) => {
      if (shouldFetchAllPosts(getState())) {
        dispatch(appActions.startRequest());
        return get(url.getPostList()).then(data => {
          dispatch(appActions.finishRequest());
          if (!data.error) {
            const { posts, postsIds, authors } = convertPostsToPlain(data);
            dispatch(fetchAllPostsSuccess(posts, postsIds, authors));
          } else {
            dispatch(appActions.setError(data.error));
          }
        });
      }
    };
  },
  // 获取帖子详情
  fetchPost: id => {
    return (dispatch, getState) => {
      if (shouldFetchPost(id, getState())) {
        dispatch(appActions.startRequest());
```

```
          return get(url.getPostById(id)).then(data => {
            dispatch(appActions.finishRequest());
            if (!data.error && data.length === 1) {
              const { post, author } = convertSinglePostToPlain(data[0]);
              dispatch(fetchPostSuccess(post, author));
            } else {
              dispatch(appActions.setError(data.error));
            }
          });
        }
      };
    },
    // 新建帖子
    createPost: (title, content) => {
      return (dispatch, getState) => {
        const state = getState();
        const author = state.auth.userId;
        const params = {
          author,
          title,
          content,
          vote: 0
        };
        dispatch(appActions.startRequest());
        return post(url.createPost(), params).then(data => {
          dispatch(appActions.finishRequest());
          if (!data.error) {
            dispatch(createPostSuccess(data));
          } else {
            dispatch(appActions.setError(data.error));
          }
        });
      };
    },
    // 更新帖子
    updatePost: (id, post) => {
      return dispatch => {
        dispatch(appActions.startRequest());
        return put(url.updatePost(id), post).then(data => {
          dispatch(appActions.finishRequest());
          if (!data.error) {
            dispatch(updatePostSuccess(data));
          } else {
```

```
          dispatch(appActions.setError(data.error));
        }
      });
    };
  }
};

// 获取帖子列表成功
const fetchAllPostsSuccess = (posts, postIds, authors) => ({
  type: types.FETCH_ALL_POSTS,
  posts,
  postIds,
  users: authors
});
// 获取帖子详情成功
const fetchPostSuccess = (post, author) => ({
  type: types.FETCH_POST,
  post,
  user: author
});
// 新建帖子成功
const createPostSuccess = post => ({
  type: types.CREATE_POST,
  post: post
});
// 更新帖子成功
const updatePostSuccess = post => ({
  type: types.UPDATE_POST,
  post: post
});
```

这里有几个地方需要注意：

（1）每一个 action type 实际上对应两个 action creator，一个创建异步 action 发送 API 请求，例如 fetchAllPosts；另一个根据 API 返回的数据创建普通的 action，例如 fetchAllPostsSuccess。

（2）供外部使用的 action creators 定义在常量对象 actions 中，例如 fetchAllPosts、fetchPost、createPost 和 updatePost。仅在模块内部使用的 action creators 不需要被导出，例如 fetchAllPostsSuccess、fetchPostSuccess、createPostSuccess 和 updatePostSuccess。

（3）Redux 的缓存作用。fetchAllPosts 中调用了 shouldFetchAllPosts，用于判断当前的 state 中是否已经有帖子列表数据，如果没有才会发送 API 请求。之所以可以这么处理，正是基于 Redux 使用一个全局 state 管理应用状态，这种缓存机制可以提高应用的性能。fetchPost 中调用的 shouldFetchPost 也是同样的作用。

（4）API 返回的数据结构往往有嵌套，我们需要把嵌套的数据结构转换成扁平的结构，这样

才能方便地被扁平化的 state 所使用。fetchAllPosts 中的 convertPostsToPlain 和 fetchPost 中 convertSinglePostToPlain 两个函数就用于执行这个转换过程的。转换过程的实现依赖于 API 返回的数据结构和业务逻辑，具体代码参考项目源代码。另外，还可以使用 normalizr（https://github.com/paularmstrong/normalizr）这个库将嵌套的数据结构转换成扁平结构。

posts 模块的 state 又拆分成 allIds 和 byId 两个子 state，每个子 state 使用一个 reducer 处理，最后通过 Redux 提供的 combineReducers 把两个 reducer 合并成一个。posts 模块的 reducer 定义如下：

```
// reducers
const allIds = (state = initialState.allIds, action) => {
  switch (action.type) {
    case types.FETCH_ALL_POSTS:
      return action.postIds;
    case types.CREATE_POST:
      return [action.post.id, ...state];
    default:
      return state;
  }
};

const byId = (state = initialState.byId, action) => {
  switch (action.type) {
    case types.FETCH_ALL_POSTS:
      return action.posts;
    case types.FETCH_POST:
    case types.CREATE_POST:
    case types.UPDATE_POST:
      return {
        ...state,
        [action.post.id]: action.post
      };
    default:
      return state;
  }
};

const reducer = combineReducers({
  allIds,
  byId
});

export default reducer;
```

至此，我们完成了 posts 模块的设计，这也是所有模块中最复杂的一个模块。posts.js 的完整代码可参考项目源代码。

9.3.4 comments 模块

comments 模块负责获取帖子的评论列表和创建新评论，与 posts 模块功能很相近，这里不再详细分析，只给出主要逻辑代码：

```
const initialState = {
  byPost: {},
  byId: {}
};

// action types
export const types = {
  FETCH_COMMENTS: "COMMENTS/FETCH_COMMENTS",     // 获取评论列表
  CREATE_COMMENT: "COMMENTS/CREATE_COMMENT"      // 新建评论
};

// action creators
export const actions = {
  // 获取评论列表
  fetchComments: postId => {
    return (dispatch, getState) => {
      if (shouldFetchComments(postId, getState())) {
        dispatch(appActions.startRequest());
        return get(url.getCommentList(postId)).then(data => {
          dispatch(appActions.finishRequest());
          if (!data.error) {
            const { comments, commentIds, users } =
convertToPlainStructure(data);
            dispatch(fetchCommentsSuccess(postId, commentIds, comments,
users));
          } else {
            dispatch(appActions.setError(data.error));
          }
        });
      }
    };
  },
  // 新建评论
  createComment: comment => {
    return dispatch => {
      dispatch(appActions.startRequest());
      return post(url.createComment(), comment).then(data => {
        dispatch(appActions.finishRequest());
```

```
          if (!data.error) {
            dispatch(createCommentSuccess(data.post, data));
          } else {
            dispatch(appActions.setError(data.error));
          }
        });
      };
    }
  };

  // 获取评论列表成功
  const fetchCommentsSuccess = (postId, commentIds, comments, users) => ({
    type: types.FETCH_COMMENTS,
    postId,
    commentIds,
    comments,
    users
  });

  // 新建评论成功
  const createCommentSuccess = (postId, comment) => ({
    type: types.CREATE_COMMENT,
    postId,
    comment
  });


  // reducers
  const byPost = (state = initialState.byPost, action) => {
    switch (action.type) {
      case types.FETCH_COMMENTS:
        return { ...state, [action.postId]: action.commentIds };
      case types.CREATE_COMMENT:
        return {
          ...state,
          [action.postId]: [action.comment.id, ...state[action.postId]]
        };
      default:
        return state;
    }
  };

  const byId = (state = initialState.byId, action) => {
```

```
  switch (action.type) {
    case types.FETCH_COMMENTS:
      return { ...state, ...action.comments };
    case types.CREATE_COMMENT:
      return { ...state, [action.comment.id]: action.comment };
    default:
      return state;
  }
};

const reducer = combineReducers({
  byPost,
  byId
});

export default reducer;
```

9.3.5　users 模块

users 模块负责维护用户信息。这个模块有些特殊，因为它不需要定义 action types 和 action creators，它响应的 action 都来自 posts 模块和 comments 模块。例如，当 posts 模块获取帖子列表数据时，users 模块也需要把帖子列表数据中的用户（作者）信息保存到自身 state 中。users.js 的主要代码如下：

```
import { types as commentTypes } from "./comments";
import { types as postTypes } from "./posts";

const initialState = {};

// reducers
const reducer = (state = initialState, action) => {
  switch (action.type) {
    // 获取评论列表和帖子列表时，更新列表数据中包含的所有作者信息
    case commentTypes.FETCH_REMARKS:
    case postTypes.FETCH_ALL_POSTS:
      return { ...state, ...action.users };
    // 获取帖子详情时，只需更新当前帖子的作者信息
    case postTypes.FETCH_POST:
      return { ...state, [action.user.id]: action.user };
    default:
      return state;
  }
};

export default reducer;
```



action 和 reducer 之间并不存在一对一的关系。一个 action 是可以被多个模块的 reducer 处理的，尤其是当模块之间存在关联关系时，这种场景更为常见。

9.3.6 ui 模块

ui 模块的功能很简单，这里只给出主要代码：

```
import { types as postTypes } from "./posts";

const initialState = {
  addDialogOpen: false,
  editDialogOpen: false
};

// action types
export const types = {
  OPEN_ADD_DIALOG: "UI/OPEN_ADD_DIALOG",    // 打开新建帖子状态
  CLOSE_ADD_DIALOG: "UI/CLOSE_ADD_DIALOG",  // 关闭新建帖子状态
  OPEN_EDIT_DIALOG: "UI/OPEN_EDIT_DIALOG",  // 打开编辑帖子状态
  CLOSE_EDIT_DIALOG: "UI/CLOSE_EDIT_DIALOG" // 关闭编辑帖子状态
};

// action creators
export const actions = {
  // 打开新建帖子的编辑框
  openAddDialog: () => ({
    type: types.OPEN_ADD_DIALOG
  }),
  // 关闭新建帖子的编辑框
  closeAddDialog: () => ({
    type: types.CLOSE_ADD_DIALOG
  }),
  // 打开编辑帖子的编辑框
  openEditDialog: () => ({
    type: types.OPEN_EDIT_DIALOG
  }),
  // 关闭编辑帖子的编辑框
  closeEditDialog: () => ({
    type: types.CLOSE_EDIT_DIALOG
  })
};

// reducers
```

```
const reducer = (state = initialState, action) => {
  switch (action.type) {
    case types.OPEN_ADD_DIALOG:
      return { ...state, addDialogOpen: true };
    case types.CLOSE_ADD_DIALOG:
    case postTypes.CREATE_POST:
      return { ...state, addDialogOpen: false };
    case types.OPEN_EDIT_DIALOG:
      return { ...state, editDialogOpen: true };
    case types.CLOSE_EDIT_DIALOG:
    case postTypes.UPDATE_POST:
      return { ...state, editDialogOpen: false };
    default:
      return state;
  }
};

export default reducer;
```

9.6.7　index 模块

在 redux/modules 路径下，我们还会创建一个 index.js，作为 Redux 的根模块。在 index.js 中做的事情很简单，只是将其余模块中的 reducer 合并成一个根 reducer。index.js 的代码如下：

```
import { combineReducers } from "redux";
import app from "./app";
import auth from "./auth";
import ui from "./ui";
import comments from "./comments";
import posts from "./posts";
import users from "./users";

// 合并所有模块的 reducer 成一个根 reducer
const rootReducer = combineReducers({
  app,
  auth,
  ui,
  posts,
  comments,
  users
});

export default rootReducer;
```

9.4 连接 Redux

Redux 模块准备好了，下面就可以通过 Redux 的 connect 函数把组件和 Redux 的 store 进行连接了。我们以组件 PostList 为例介绍连接过程。

9.4.1 注入 state

PostList 组件需要从 Redux 的 store 中获取以下数据：当前登录用户、帖子列表数据和新建帖子编辑框的 UI 状态，根据 store 中 state 的结构，一种最直接的获取所需数据的方式如下：

```
// containers/PostList/index.js

const getPostList = state => {
  return state.posts.allIds.map(id => {
    return state.posts.byId[id];
  });
};

const mapStateToProps = state => {
  return {
    user: state.auth,      //当前登录用户
    posts: getPostList(state),  //帖子列表数据
    isAddDialogOpen: state.ui.addDialogOpen   //新建帖子编辑框的 UI 状态
  };
};
```

user 和 isAddDialogOpen 两个属性可以直接从 state 中获取，但这种获取方式意味着组件必须了解 state 的结构，而且 state 结构发生变化时，组件也必须通过新的 state 结构访问使用的属性。总之，container 层和 Redux 的 module 层有了强耦合。良好的模块设计对外暴露的应该是模块的接口，而不是模块的具体结构。我们可以利用 Redux 中的 selector 解决这个问题。selector 是一个函数，用于从 state 中获取外部组件所需的数据。这样，当组件需要使用 state 中的数据时，不再直接访问 state，而是通过 selector 获取。上面示例中的 getPostList 就是一个 selector，但 selector 适合定义在相关 Redux 模块中，即一个 Redux 模块不仅包含 action types、action creators 和 reducers，还包含从该模块 state 中获取数据的 selectors。

我们在 auth 模块中定义获取当前用户的 selector：

```
// redux/modules/auth.js

// selectors
export const getLoggedUser = state => state.auth;
```

在 ui 模块中定义获取新建帖子编辑框的 UI 状态的 selector：

```
// redux/modules/ui.js

// selectors
export const isAddDialogOpen = state => {
  return state.ui.addDialogOpen;
};
```

当需要以多个模块的 state 作为 selector 的输入时，这个 selector 就不再适合定义在某个具体模块中，这种情况下，我们定义到 redux/module/index.js 中。例如，posts 模块的 state 只包含作者的 ID 信息，但当展示帖子列表时，需要显示的是作者的用户名，而作者信息需要从 users 模块获取，因此获取帖子列表的 selector 就应该定义在 redux/module/index.js 中，代码如下：

```
// redux/modules/index.js
import { getPostIds, getPostById } from "./posts";
import { getUserById } from "./users";

export const getPostListWithAuthors = state => {
  // 通过 posts 模块的 getPostIds 获取所有帖子的 id
  const postIds = getPostIds(state);
  return postIds.map(id => {
    // 通过 posts 模块的 getPostById 获取每个帖子的详情
    const post = getPostById(state, id);
    // users 模块的 getUserById 获取作者信息，并将作者信息合并到 post 对象中
    return { ...post, author: getUserById(state, post.author) };
  });
};
```

注意，getPostListWithAuthors 中还使用到了 posts 模块和 users 模块的 selectors。这样通过 selector 进行一些逻辑的处理和数据结构的转换，容器组件可以更加便利地使用全局 state 中的数据。最终，PostList 注入 state 的代码如下：

```
// containers/PostList/index.js
import { getLoggedUser } from "../../redux/modules/auth";
import { isAddDialogOpen } from "../../redux/modules/ui";
import { getPostListWithAuthors } from "../../redux/modules";

const mapStateToProps = (state, props) => {
  return {
    user: getLoggedUser(state),
    posts: getPostListWithAuthors(state),
    isAddDialogOpen: isAddDialogOpen(state)
  };
};
```

9.4.2 注入 action creators

接下来为 PostList 注入使用到的 action creators。PostList 中需要获取帖子列表、新建帖子，还需要控制新建帖子编辑框的 UI 状态，因此，PostList 需要使用到 posts 和 ui 两个模块中的 action creators，代码如下：

```
// containers/PostList/index.js
import { bindActionCreators } from "redux";
import { actions as postActions } from "../../redux/modules/posts";
import { actions as uiActions } from "../../redux/modules/ui";

const mapDispatchToProps = dispatch => {
  return {
    ...bindActionCreators(postActions, dispatch),
    ...bindActionCreators(uiActions, dispatch)
  };
};
```

其中，bindActionCreators 是 Redux 提供的一个工具函数，它使用 store 的 dispatch 方法把参数对象中包含的每个 action creator 包裹起来，这样就不需要显式地使用 dispatch 方法去发送 action 了，而是可以直接调用 action creator 函数（bindActionCreators 返回的对象的属性就是可以直接调用的 action creator）。例如，不使用 bindActionCreators 时，有一个如下定义的 mapDispatchToProps：

```
const mapDispatchToProps = dispatch => {
  return {
    someActionCreator: someActionCreator,
  };
};
```

在组件中发起对应的 action 需要这样调用：

```
this.props.dispatch(this.props.someActionCreator());
```

mapDispatchToProps 使用 bindActionCreators 后：

```
const mapDispatchToProps = dispatch => {
  return {
    someActionCreator: bindActionCreators(someActionCreator, dispatch),
  };
};
```

只需要这样调用即可发送 action：

```
this.props.someActionCreator();
```

注意，bindActionCreators 的第一个参数可以是一个函数或者一个普通对象，如果是函数类型，这个函数就是一个 action creator；如果是普通对象类型，对象的每一个属性就是一个 action creator。

9.4.3　connect 连接 PostList 和 Redux

最后，利用 Redux 的 connect 函数将 PostList 和 Redux 连接起来，并导出连接后的组件：

```
// containers/PostList/index.js
import { connect } from "react-redux";

export default connect(mapStateToProps, mapDispatchToProps)(PostList);
```

完整的 containers/PostList/index.js 代码如下：

```
import React, { Component } from "react";
import { bindActionCreators } from "redux";
import { connect } from "react-redux";
import PostsView from "./components/PostsView";
import PostEditor from "../Post/components/PostEditor";
import { getLoggedUser } from "../../redux/modules/auth";
import { actions as postActions } from "../../redux/modules/posts";
import { actions as uiActions, isAddDialogOpen } from "../../redux/
modules/ui";
import { getPostListWithAuthors } from "../../redux/modules";
import "./style.css";

class PostList extends Component {
  componentDidMount() {
    this.props.fetchAllPosts();  // 获取帖子列表
  }

  // 保存帖子
  handleSave = data => {
    this.props.createPost(data.title, data.content);
  };

  // 取消新建帖子
  handleCancel = () => {
    this.props.closeAddDialog();
  };

  // 新建帖子
  handleNewPost = () => {
    this.props.openAddDialog();
  };

  render() {
    const { posts, user, isAddDialogOpen } = this.props;
```

```
      return (
        <div className="postList">
          <div>
            <h2>帖子列表</h2>
            {user.userId ? (
              <button onClick={this.handleNewPost}>发帖</button>
            ) : null}
          </div>
          {isAddDialogOpen ? (
            <PostEditor onSave={this.handleSave} onCancel={this.handleCancel} />
          ) : null}
          <PostsView posts={posts}/>
        </div>
      );
    }
  }

const mapStateToProps = (state, props) => {
  return {
    user: getLoggedUser(state),
    posts: getPostListWithAuthors(state),
    isAddDialogOpen: isAddDialogOpen(state)
  };
};

const mapDispatchToProps = dispatch => {
  return {
    ...bindActionCreators(postActions, dispatch),
    ...bindActionCreators(uiActions, dispatch)
  };
};

export default connect(mapStateToProps, mapDispatchToProps)(PostList);
```

其他容器组件和 Redux 的连接方式与 PostList 相同，区别只是注入的 state 和 action creators 不同。限于篇幅，这里不再一一介绍。

最后，我们还需要把 Redux 的 store 通过 Provider 组件注入应用中，这个操作在应用的根组件中完成：

```
// index.js

import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
```

```
import configureStore from "./redux/configureStore";
import App from "./containers/App";
// 创建 store
const store = configureStore();

ReactDOM.render(
  {/* 通过 Provider 注入 store */}
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

大功告成！完整的代码可参考源代码。

9.5　Redux 调试工具

Redux DevTools 是一款用于调试 Redux 应用的浏览器插件，它可以实时地显示当前应用的 state 信息、action 触发的记录以及 state 的变化，在开发过程中非常有用。目前，这个插件支持 Chrome 和 FireFox 浏览器。以 Chrome 浏览器为例，可以在 Chrome 的应用商店中下载安装 Redux DevTools 插件。然后在 configureStore.js 中，使用 Redux DevTools 创建“加强版”的 store：

```
import { createStore, applyMiddleware, compose } from "redux";
import thunk from "redux-thunk";
import rootReducer from "./modules";

let finalCreateStore;
// 如果程序运行在非生产模式下，且浏览器安装了调试插件，则创建包含调试插件的 store
if (process.env.NODE_ENV !== "production" &&
window.__REDUX_DEVTOOLS_EXTENSION__) {
  finalCreateStore = compose(
    applyMiddleware(thunk),
    window.__REDUX_DEVTOOLS_EXTENSION__()
  )(createStore);
} else {
  finalCreateStore = applyMiddleware(thunk)(createStore);
}

export default function configureStore(initialState) {
  const store = finalCreateStore(rootReducer, initialState);
  return store;
}
```

注意，我们只在开发环境下启用 Redux DevTools。运行程序，打开 Chrome 浏览器的开发者工具窗口，选择 Redux 标签页，即可看到 Redux DevTools 的效果，如图 9-3 所示。

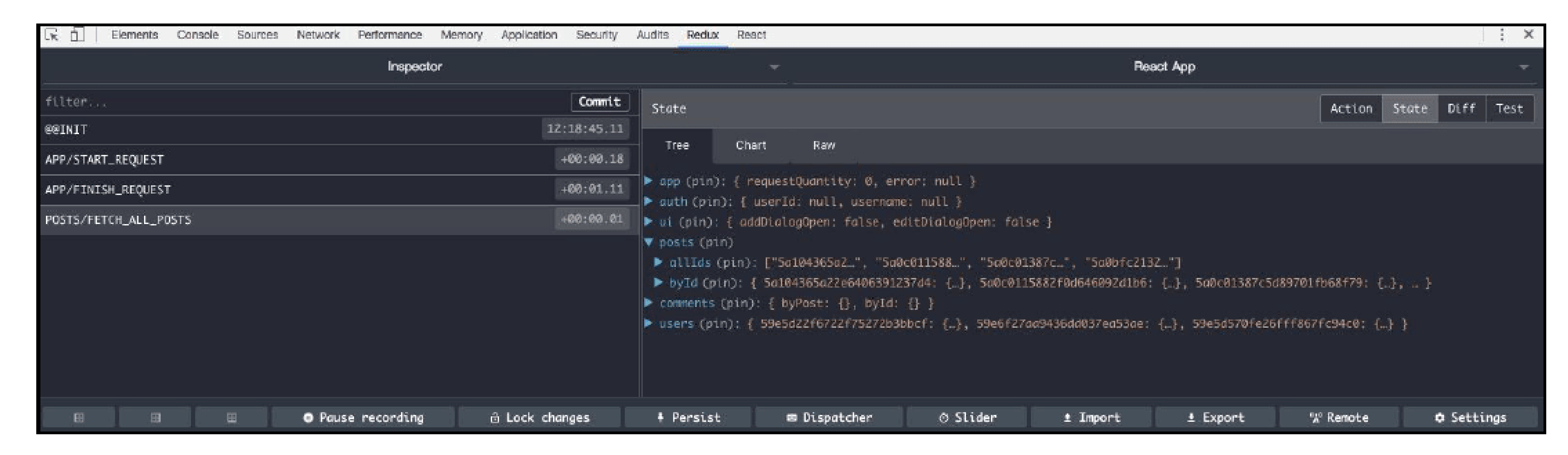


图 9-3

除了安装浏览器插件的方式外，还可以直接在项目中集成 Redux 的调试工具，具体方式可参考 https://www.npmjs.com/package/redux-devtools，这里不再展开介绍。

9.6　性能优化

Redux 使数据流动变得清晰，但目前我们的项目中还存在一些不必要的状态重复计算和 UI 重复渲染。下面将对程序性能做进一步优化。

9.6.1　React Router 引起的组件重复渲染问题

Redux 和 React Router 集成使用时，容易碰到一个很隐蔽的性能问题。先来看一下组件 app 的代码：

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch } from "react-router-dom";
import { bindActionCreators } from "redux";
import { connect } from "react-redux";
import asyncComponent from "../../utils/AsyncComponent";
import ModalDialog from "../../components/ModalDialog";
import Loading from "../../components/Loading";
import { actions as appActions, getError, getRequestQuantity } from "../../redux/modules/app";

// 异步加载 Home 组件
const AsyncHome = asyncComponent(() => import("../Home"));
// 异步加载 Login 组件
const AsyncLogin = asyncComponent(() => import("../Login"));

class App extends Component {
```

```
  render() {
    const { error, requestQuantity } = this.props;
    const errorDialog = error && (
      <ModalDialog onClose={this.props.removeError}>
        {error.message || error}
      </ModalDialog>
    );

    return (
      <div>
        <Router>
          <Switch>
            <Route exact path="/" component={AsyncHome} />
            <Route path="/login" component={AsyncLogin} />
            <Route path="/posts" component={AsyncHome} />
          </Switch>
        </Router>
        {errorDialog}
        {requestQuantity > 0 && <Loading />}
      </div>
    );
  }
}

const mapStateToProps = (state, props) => {
  return {
    error: getError(state),
    requestQuantity: getRequestQuantity(state)
  };
};

const mapDispatchToProps = dispatch => {
  return {
    ...bindActionCreators(appActions, dispatch)
  };
};

export default connect(mapStateToProps, mapDispatchToProps)(App);
```

app 中既使用到 React Router 的 Route，又使用到从 Redux store 中获取的 error 和 requestQuantity 两个状态。帖子列表组件 PostList 在发送获取帖子列表数据的 action 时，会改变 requestQuantity 的值，这时候 app 的 render 方法会再次被调用，这是正常的情况，但 Route 中定义的组件（比如 Home，实际上是组件 AsyncHome，AsyncHome 中渲染了组件 Home）的 render 方法也会被再次调用。再来看一下组件 Home 的代码：

```
import React, { Component } from "react";
import { Route } from "react-router-dom";
import { bindActionCreators } from "redux";
import { connect } from "react-redux";
import Header from "../../components/Header";
import asyncComponent from "../../utils/AsyncComponent";
import { actions as authActions, getLoggedUser } from "../../redux/modules/
auth";

// 异步加载 Post 组件
const AsyncPost = asyncComponent(() => import("../Post"));
// 异步加载 PostList 组件
const AsyncPostList = asyncComponent(() => import("../PostList"));

class Home extends Component {
  // 省略其余代码
}

const mapStateToProps = (state, props) => {
  return {
    user: getLoggedUser(state)
  };
};

const mapDispatchToProps = dispatch => {
  return {
    ...bindActionCreators(authActions, dispatch)
  };
};

export default connect(mapStateToProps, mapDispatchToProps)(Home);
```

Home 也是一个容器组件，它使用到 Redux store 中的状态 user，当 requestQuantity 发生变化时，从 store 中获取的 user 还是原来的对象，也就是说 Home 的 mapStateToProps 新返回的对象和之前的对象符合浅比较（shallow comparison）相等的条件，根据 8.3.4 小节的介绍，Home 组件的 render 方法不应该被再次调用。

这个问题的根结在于 React Router 的 Route 组件。下面是 Route 的部分源码（注意注释部分）：

```
componentWillReceiveProps(nextProps, nextContext) {
  // 省略部分代码

  // 注意这里，computeMatch 每次返回的都是一个新对象，如此一来，每次 Route 更新，
  //componentWillReceiveProps 被调用，setState 都会重新设置一个新的 match 对象
  this.setState({
```

```
      match: this.computeMatch(nextProps, nextContext.router)
    })
  }

  render() {
    const { match } = this.state
    const { children, component, render } = this.props
    const { history, route, staticContext } = this.context.router
    const location = this.props.location || route.location
    // 注意这里，这是传递给 Route 中的组件的属性
    const props = { match, location, history, staticContext }

    if (component)
      return match ? React.createElement(component, props) : null

    if (render)
      return match ? render(props) : null

    if (typeof children === 'function')
      return children(props)

    if (children && !isEmptyChildren(children))
      return React.Children.only(children)

    return null
  }
```

app render 方法的执行会导致 Route 的 componentWillReceiveProps 执行，componentWillReceiveProps 每次都会调用 setState 设置 match，match 由 computeMatch 计算而来，computeMatch 每次都会返回一个新的对象。这样，每次 Route 更新（componentWillReceiveProps 被调用）都将创建一个新的 match，而这个 match 又会作为 props 传递给 Route 中定义的组件。于是，Home 组件在更新阶段总会收到一个新的 match 属性，react-redux 既会比较组件依赖的 state 的变化，又会比较组件接收的 props 的变化，这种情况下，props 总是改变的，组件的 render 方法被重新调用。事实上，在上面的情况中，Route 传递给 Home 的其他属性 location、history、staticContext 都没有改变，match 虽然是一个新对象，但对象的内容并没有改变（一直处在同一页面，URL 并没有发生变化，match 的计算结果自然也没有变）。

我们可以通过创建一个高阶组件在高阶组件内重写组件的 shouldComponentUpdate 方法，如果 Route 传递的 location 属性没有发生变化（表示处于同一页面），就返回 false，从而阻止组件继续更新。然后使用这个高阶组件包裹每一个要在 Route 中使用的组件。

新建一个高阶组件 connectRoute：

```
import React from "react";
```

```
export default function connectRoute(WrappedComponent) {
  return class extends React.Component {
    shouldComponentUpdate(nextProps) {
      return nextProps.location !== this.props.location;
    }

    render() {
      return <WrappedComponent {...this.props} />;
    }
  };
}
```

用 connectRoute 包裹 Home、Login：

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch } from "react-router-dom";
import { bindActionCreators } from "redux";
import { connect } from "react-redux";
import asyncComponent from "../../utils/AsyncComponent";
import ModalDialog from "../../components/ModalDialog";
import Loading from "../../components/Loading";
import { actions as appActions, getError, getRequestQuantity } from
"../../redux/modules/app";
import connectRoute from "../../utils/connectRoute";

// connectRoute 包裹 Home
const AsyncHome = connectRoute(asyncComponent(() => import("../Home")));
// connectRoute 包裹 Login
const AsyncLogin = connectRoute(asyncComponent(() => import("../Login")));

class App extends Component {
  // ...
}

const mapStateToProps = (state, props) => {
  return {
    error: getError(state),
    requestQuantity: getRequestQuantity(state)
  };
};

const mapDispatchToProps = dispatch => {
  return {
    ...bindActionCreators(appActions, dispatch)
```

```
  };
};

export default connect(mapStateToProps, mapDispatchToProps)(App);
```

这样，app 依赖的 store 中的 state 改变就不会再导致 Route 内组件的 render 方法的重新执行了。其他使用到 Route 的容器组件中也需要做同样的处理。

我们再来思考一种场景，如果 app 使用的 store 中的 state 同样会影响到 Route 的属性，比如 requestQuantity 大于 0 时，Route 的 path 会改变，假设变成<Route path="/home/fetching" component={AsyncHome} />，而 Home 内部假设又需要用到 Route 传递的 path（通过 props.match.path 获取），这时候就需要 Home 组件重新 render。但因为在高阶组件 connectRoute 的 shouldComponentUpdate 中，我们只是根据 location 做判断，此时的 location 依然没有发生变化，导致 Home 并不会重新渲染。这是一种很特殊的场景，但是想通过这种场景告诉大家，高阶组件 connectRoute 中 shouldComponentUpdate 的判断条件需要根据实际业务场景做决策。绝大部分场景下，上面的高阶组件是足够使用的。

另外，React Router 的这个问题并不是只和 Redux 一起使用时才会遇到，当和 MobX 一起使用或 Route 内使用的组件继承自 React 的 PureComponent 时，也存在同样的问题。

9.6.2　Immutable.JS

Redux 的 state 必须是不可变对象，reducer 中每次返回的 state 都是一个新对象。为了保证这一点，我们需要写一些额外的处理逻辑，例如使用 Object.assign 方法或 ES6 的扩展运算符（...）创建新的 state 对象。

Immutable.JS 的作用在于以更加高效的方式创建不可变对象，主要优点有 3 个：保证数据的不可变、丰富的 API 和优异的性能。

1. 保证数据的不可变

通过 Immutable.JS 创建的对象在任何情况下都无法被修改，这样就可以防止由于开发者的粗心大意导致直接修改了 Redux 的 state。

2. 丰富的 API

Immutable.JS 提供了丰富的 API 创建不同类型的不可变对象，如 Map、List、Set、Record 等，它还提供了大量 API 用于操作这些不可变对象，如 get、set、sort、filter 等。

3. 优异的性能

一般情况下，使用不可变对象会涉及大量的复制操作，给程序性能带来影响。Immutable.JS 在这方面做了大量优化，将使用不可变对象带来的性能损耗降低到可以忽略不计。

使用 Immutable.JS 前，需要先在项目根路径下安装这个依赖：

```
npm install immutable
```

然后，就可以在项目中使用 Immutable.JS 了，下面是一个简单示例：

```
import Immutable from "immutable";

const map1 = Immutable.Map({ a: 1, b: 2, c: 3 });
const map2 = map1.set('b', 50);
const map3 = map1.merge({b: 100});
map1.get('b') + " vs. " + map2.get('b') + " vs. " + map3.get('b');  // 2 vs. 
50 vs. 100
```

我们先使用 Immutable.JS 的 Map API 创建了一个 Map 结构的不可变对象 map1，然后分别使用 set、merge 方法修改 map1，最后通过 get 方法从不可变对象中取出 b 的值，输出表明，set、merge 并没有修改原有的 map1 对象，而是创建了一个新的对象。这就是 Immutable.JS 的最大特征，对象一旦创建，就无法再次修改。关于 Immutable.JS 完整的 API 介绍可参考官方文档：https://facebook.github.io/immutable-js/。

当 Immutable.JS 和 Redux 一起使用时，需要通过 Immutable.JS 的 API 创建 Redux 的全局 state，reducer 中通过 Immutable.JS 的 API 修改 state。我们以 BBS 项目中的 posts 模块为例详细介绍如何引入 Immutable.JS。

（1）用 Immutable.JS 创建模块使用的初始 state：

```
import Immutable from "immutable";

const initialState = Immutable.fromJS({
  allIds: [],
  byId: {}
});
```

posts 模块中的每一个 reducer 设置 state 默认值的方式也需要修改：

```
const allIds = (state = initialState.get("allIds"), action) => {
  // ...
};

const byId = (state = initialState.get("byId"), action) => {
  // ...
};
```

（2）当 reducer 接收到 action 时，reducer 内部也需要通过 Immutable.JS 的 API 来修改 state，代码如下：

```
const allIds = (state = initialState.get("allIds"), action) => {
  switch (action.type) {
    case types.FETCH_ALL_POSTS:
      // Immutable.List 创建一个 List 类型的不可变对象
      return Immutable.List(action.postIds);
    case types.CREATE_POST:
      // 使用 unshift 向 List 类型的 state 增加元素
```

```
      return state.unshift(action.post.id);
    default:
      return state;
  }
};

const byId = (state = initialState.get("byId"), action) => {
  switch (action.type) {
    case types.FETCH_ALL_POSTS:
      // 使用 merge 合并获取到的 post 列表数据
      return state.merge(action.posts);
    case types.FETCH_POST:
    case types.CREATE_POST:
    case types.UPDATE_POST:
      // 使用 merge 修改对应 post 的数据
      return state.merge({ [action.post.id]: action.post });
    default:
      return state;
  }
};
```

（3）引入 redux-immutable。之前使用 Redux 提供的 combineReducers 函数合并 reducer，但 combineReducers 只能识别普通 JavaScript 对象组成的 state，无法识别 Immutable.JS 创建的对象组成的 state。我们可以使用 redux-immutable 这个库提供的 combineReducers 解决这个问题。先在项目中安装 redux-immutable：

```
npm install redux-immutable
```

使用 redux-immutable 的 combineReducers 合并 reducer:

```
import { combineReducers } from "redux-immutable";

const reducer = combineReducers({
  allIds,
  byId
});

export default reducer;
```

（4）修改 selectors，让 selectors 返回 Immutable.JS 类型的不可变对象：

```
// selectors
export const getPostIds = state => state.getIn(["posts", "allIds"]);

export const getPostList = state => state.getIn(["posts", "byId"]);
```

```
export const getPostById = (state, id) => state.getIn(["posts", "byId", id]);
```

修改 selectors 时，不要忘记 redux/module/index.js 中定义的复杂 selectors 同样需要修改，因为这部分修改相对复杂些，这里给出修改后的代码：

```
import { getCommentIdsByPost, getCommentById } from "./comments";
import { getPostIds, getPostById } from "./posts";
import { getUserById } from "./users";

// 获取包含完整作者信息的帖子列表
export const getPostListWithAuthors = state => {
  const allIds = getPostIds(state);
  return allIds.map(id => {
    const post = getPostById(state, id);
    return post.merge({ author: getUserById(state, post.get("author")) });
  });
};

// 获取帖子详情
export const getPostDetail = (state, id) => {
  const post = getPostById(state, id);
  return post
    ? post.merge({ author: getUserById(state, post.get("author")) })
    : null;
};

// 获取包含完整作者信息的评论列表
export const getCommentsWithAuthors = (state, postId) => {
  const commentIds = getCommentIdsByPost(state, postId);
  if (commentIds) {
    return commentIds.map(id => {
      const comment = getCommentById(state, id);
      return comment.merge({ author: getUserById(state,
comment.get("author")) });
    });
  } else {
    return Immutable.List();
  }
};
```

这里省略了 comments 和 users 模块中新版本 selectors 的定义。

（5）在容器组件 PostList 中使用新版本的 selectors：

```
import { getLoggedUser } from "../../redux/modules/auth";
import { isAddDialogOpen } from "../../redux/modules/ui";
```

```
import { getPostListWithAuthors } from "../../redux/modules";

const mapStateToProps = (state, props) => {
  return {
    user: getLoggedUser(state),
    posts: getPostListWithAuthors(state),
    isAddDialogOpen: isAddDialogOpen(state)
  };
};
```

因为 mapStateToProps 返回的对象的属性是 Immutable.JS 类型的不可变对象，所以在容器组件中使用时，也需要通过 Immutable.JS 的 API 获取不可变对象中的属性值。但展示组件应该是对 Immutable.JS 的使用无感知的，也就是容器组件需要把 Immutable.JS 类型的不可变对象转换成普通 JavaScript 对象后，再传递给展示组件使用（否则展示组件复用时，还必须捆绑 Immutable.JS）。主要的变化发生在 containers/PostList/index.js 的 render 方法中：

```
  render() {
    const { posts, user, isAddDialogOpen } = this.props;
    const rawPosts = posts.toJS(); // 转换成普通 JavaScript 对象
    return (
      <div className="postList">
        <div>
          <h2>帖子列表</h2>
          {user.get("userId") ? (
            <button onClick={this.handleNewPost}>发帖</button>
          ) : null}
        </div>
        {isAddDialogOpen ? (
          <PostEditor onSave={this.handleSave} onCancel={this.handleCancel} />
        ) : null}
        <ul>
          {rawPosts.map(item => (
            <Link key={item.id} to={`/posts/${item.id}`}>
              <PostItem post={item} />
            </Link>
          ))}
        </ul>
      </div>
    );
  }
```

不可变对象 posts 先通过 toJS 方法转换成普通的 JavaScript 对象，然后才提供给展示组件 PostItem 使用。



注　意

不要在 mapStateToProps 中使用 toJS()将 Immutable.JS 类型的不可变对象转换成普通 JavaScript 对象。因为 toJS()每次返回的都是一个新对象，这将导致 Redux 每次使用浅比较判断 mapStateToProps 返回的对象是否改变时，都认为发生了修改，从而导致不必要的重复调用组件的 render 方法。

至此，我们就完成了 posts 模块使用 Immutable.JS 的重构，其他模块的修改可参考源代码：/chapter-09/bbs-redux-immutable。

通过这些修改可以发现使用 Immutable.JS 也有一些缺点，例如，Immutable.JS 创建的对象难以和普通 JavaScript 对象混合使用；操作 Immutable.JS 对象也不是很便捷，必须使用其提供的 API 完成；Immutable.JS 对象由于其特殊的结构，因此调试起来也更为麻烦。当项目的 state 数据结构并不是很复杂时，使用 Immutable.JS 带来的性能提升并不显著，所以读者可根据项目实际情况有选择性地使用 Immutable.JS。

9.6.3　Reselect

我们知道，Redux state 的任意改变都会导致所有容器组件的 mapStateToProps 的重新调用，进而导致使用到的 selectors 重新计算。但 state 的一次改变只会影响到部分 selectors 的计算值，只要这个 selector 使用到的 state 的部分未发生改变，selector 的计算值就不会发生改变，理论上这部分计算时间是可以被节省的。

先来看一下之前用来获取帖子列表的 selector：

```
import { getPostIds, getPostById } from "./posts";
import { getUserById } from "./users";

export const getPostListWithAuthors = state => {
  const allIds = getPostIds(state);
  return allIds.map(id => {
    const post = getPostById(state, id);
    return post.merge({ author: getUserById(state, post.get("author")) });
  });
};
```

getPostListWithAuthors 需要根据 posts 模块和 users 模块的 state 计算出容器组件 PostList 所需格式的对象。当其他模块的 state 发生改变时，例如 ui 模块的编辑框状态发生变化，PostList 的 mapStateToProps 会被重新调用，getPostListWithAuthors 也就被重新调用，但这种场景下，getPostListWithAuthors 的计算结果并不会发生改变，完全可以不必重新计算，而是直接使用上次的计算结果即可。

Reselect 正是用来解决这类问题的。Reselect 可以创建具有记忆功能的 selectors，当 selectors 计算使用的参数未发生改变时，不会再次计算，而是直接使用上次缓存的计算结果。

现在，我们把 getPostListWithAuthors 改造成具有记忆功能的 selector。首先，需要安装 reselect 库：

```
npm install reselect
```

Reselect 提供了一个函数 createSelector 用来创建具有记忆功能的 selector。createSelector 的定义如下：

```
createSelector([inputSelectors], resultFunc)
```

它接收两个参数，第一个参数[inputSelectors]是数组类型，数组的元素是 selector，第二个参数 resultFunc 是一个转换函数，[inputSelectors]中每一个 selector 的计算结果都会作为参数传递给 resultFunc。createSelector 的返回值是一个具有记忆功能的 selector，这个 selector 每次被调用时，使用运算符（===）判断[inputSelectors]中的 selector 计算结果相较前一次是否发生变化，如果所有的 selector 计算结果都没有变化，就直接返回前一次的计算结果。改造后的 getPostListWithAuthors 如下：

```
// redux/modules/index.js
import { getPostIds, getPostList, getPostById } from "./posts";
import { getUsers } from "./users";

export const getPostListWithAuthors = createSelector(
  [getPostIds, getPostList, getUsers],
  (allIds, posts, users) => {
    return allIds.map(id => {
      let post = posts.get(id);
      return post.merge({ author: users.get(post.get("author")) });
    });
  }
);
```

现在，只要 getPostIds、getPostList 和 getUsers 的返回值不变（本质上是 posts 和 users 模块的 state 没有改变），getPostListWithAuthors 就不会重新计算。selector 的计算逻辑越复杂，Redux 全局 state 的改变频率越高，Reselect 带来的性能提升就越大。另外请注意，如果一个 selector 并不执行任何计算逻辑，只是单纯地从 state 中读取属性值，例如 posts 模块中的 getPostIds 和 getPostList，就没有必要使用 Reselect 进行改造。本节项目源代码的目录为/chapter-09/bbs-redux-reselect。

9.7　本章小结

本章结合项目实例，从 Redux 项目结构的组织方式、state 的设计、Redux 模块的设计等方面详细介绍了如何在真实项目中使用 Redux。本章还讨论了 React Router 的使用引起的组件重复渲染的问题。在性能优化方面，Immutable.JS 和 Reselect 是最常用的用于优化 Redux 项目性能的两个库。Immutable.JS 和 Reselect 虽然都能带来性能的提升，但同时也会增加一部分代码的复杂度，当程序的性能并没有问题时，尤其是考虑到 Redux 和 React 本身就已经做了大量性能优化的工作，完全可以不引入这些库。

第 10 章

MobX：简单可扩展的状态管理解决方案

MobX 是 Redux 之后的一个状态管理库，基于响应式管理状态，整体是一个观察者模式的架构，存储 state 的 store 是被观察者，使用 store 的组件是观察者。MobX 可以有多个 store 对象，store 使用的 state 也是可变对象，这些都是和 Redux 的不同点，相较于 Redux，MobX 更轻量，也受到了很多开发者的青睐。

10.1 简　介

MobX 通过函数响应式编程的思想使状态管理变得简单和可扩展。MobX 背后的哲学是：任何可以从应用程序的状态中获取/衍生的数据都应该可以自动被获取/衍生。和 Redux 一样，MobX 也是采用单向数据流管理状态：通过 action 改变应用的 state，state 的改变进而会导致受其影响的 views 更新，如图 10-1 所示。



图 10-1

MobX 包含的主要概念有 4 个：state（状态）、computed value（计算值）、reaction（响应）和 action（动作）。computed value 和 reaction 会自动根据 state 的改变做最小化的更新，并且这个更新过程是同步执行的，也就是说，action 更改 state 后，新的 state 是可以被立即获取的。注意，computed value 采用的是延迟更新，只有当 computed value 被使用时它的值才会被重新计算，当

computed value 不再被使用时（例如使用它的组件已经被卸载），它将会被自动回收。computed value 必须是纯函数，不能使用它修改 state。


注 意

一般来说，驱动应用的任何数据都可以称为 state（状态）。但 MobX 中提到的 state 实际上都是指可观测的 state，因为对于不可观测的 state，它们的修改并不会自动产生影响，对 MobX 的数据流来说是没有意义的。

MobX 中大量使用了 ES.Next 的装饰器语法，但装饰器语法目前还处于试验阶段，create-react-app 创建的项目默认是不支持的。我们先来解决这个问题再继续介绍 MobX。要支持装饰器语法，可以使用 npm run eject 命令将项目配置"弹出"，然后添加 babel-plugin-transform-decorators-legacy 这个 Babel 插件，也可以使用 custom-react-scripts（https://www.npmjs.com/package/custom-react-scripts）来创建项目。本书使用 custom-react-scripts 这种方式。具体方式为，在使用 create-react-app 创建项目时，指定--scripts-version 参数的值为 custom-react-scripts：

```
create-react-app my-app --scripts-version custom-react-scripts
```

创建的项目根路径下有一个名为.env 的文件，这个文件中定义了 custom-react-scripts 为项目新增的特性，例如支持装饰器语法、支持 Less、支持 Sass 等。打开这个文件，可以发现有一项配置是 REACT_APP_DECORATORS = true，这项配置就是用来启用装饰器语法的。

另外，很多编辑器遇到装饰器语法会提示错误，需要进行额外设置以支持装饰器语法。例如，本书使用的 VS Code 需要在项目的根路径下创建一个文件 jsconfig.json，文件的内容为：

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

现在，我们就可以在项目中随意使用装饰器语法了。

我们通过 todos 应用来简单介绍这 4 个概念。本节项目的源码目录为/chapter-10/todos-mobx。

MobX 可以使用 object、array、class 等任意数据结构定义可观测的 state。例如，使用 class 定义一个可观测的 state Todo 代表一项任务：

```
import { observable } from "mobx";

class Todo {
  id = Math.random();
  @observable title = "";
  @observable finished = false;
}
```

这里使用的@observable 就属于装饰器语法，你也可以不使用它，直接使用 MobX 提供的函数定义一个可观测的状态。例如，下面是用 ES 5 语法实现的等价代码：

```
import { extendObservable } from "mobx";

function Todo() {
  this.id = Math.random();
  extendObservable(this, {
    title: "",
    finished: false
  });
}
```

显然，使用装饰器的代码更为清晰简洁，MobX 使用了大量装饰器语法，这也是官方推荐的方式，本书也是使用装饰器语法完成 MobX 的项目代码。经过@observable 的修饰之后，Todo 的 title 和 finished 两个属性变成可观测状态（注意属性和状态的概念，状态对象的属性也是状态），它们的改变会自动被观察者获知。id 没有被@observable 修饰，所以只是一个普通属性。

基于可观测的 state 可以创建 computed value。例如，todos 中需要获取未完成的任务总数，使用@computed 定义一个 unfinishedTodoCount 的 computed value，计算未完成的任务总数：

```
import { observable, computed } from "mobx";

class TodoList {
  @observable todos = [];

  // 根据 todos 和 todo.finished 两个 state，创建 computed value
  @computed get unfinishedTodoCount() {
    return this.todos.filter(todo => !todo.finished).length;
  }
}
```

这里又定义了一个新的 state：TodoList。TodoList 的属性 todos 是一个可观测的数组，它的元素是前面定义的 Todo 的实例对象。当 todos 中的元素数量发生变化或某一个 todo 元素的 finished 属性变化时，unfinishedTodoCount 都会自动更新（更严谨的说法是，在需要时才自动更新，后面还会介绍）。

除了 computed value 会响应 state 的变化外，reaction 也会响应 state 的变化，不同的是，reaction 并不创建一个新值，而是用来执行有副作用的逻辑，例如输出日志到控制台、发送网络请求、根据 React 组件树更新 DOM 等。mobx-react 包提供了@observer 装饰器和 observer 函数，可以将 React 组件封装成 reaction，自动根据 state 的变化更新组件 UI。例如，创建 TodoListView 和 TodoView 两个组件（也是两个 reaction）代表应用的 UI：

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import { observer } from "mobx-react";
import { action } from "mobx";
```

```
// 使用@observer 装饰器创建 reaction
@observer
class TodoListView extends Component {
  render() {
    return (
      <div>
        <ul>
          {this.props.todoList.todos.map(todo => (
            <TodoView todo={todo} key={todo.id} />
          ))}
        </ul>
        Tasks left: {this.props.todoList.unfinishedTodoCount}
      </div>
    );
  }
}

// 使用 observer 函数创建 reaction
const TodoView = observer(({ todo }) => {
  return (
    <li>
      <input
        type="checkbox"
         checked={todo.finished}
      />
      {todo.title}
    </li>
  );
});

const store = new TodoList();
ReactDOM.render(<TodoListView todoList={store} />,
document.getElementById('root'));
```

TodoListView 使用到的可观测 state 是 todos 和 todo.finished（通过 unfinishedTodoCount 间接使用），因此它们的改变将会更新 TodoListView 代表的 DOM，同样地，todo.finished 和 todo.title 的改变会更新使用这个 todo 对象的 TodoView 代表的 DOM。

MobX 通过 action 改变 state。我们在 TodoView 中定义一个 action，用来改变 todo.finish：

```
const TodoView = observer(({ todo }) => {
  // 定义 action，改变 todo.finish
  const handleClick = action(() => todo.finished = !todo.finished);
  return (
```

```
        <li>
          <input
            type="checkbox"
             checked={todo.finished}
             onClick={handleClick}
          />
          {todo.title}
        </li>
      );
    });
```

handleClick 就是用来改变状态 todo.finish 的 action，一般习惯使用 MobX 提供的 action 函数包裹应用中定义的 action。至此，这个精简版的 todos 应用已经包含了 MobX 涉及的主要概念。

10.2 主要组成

本节将详细介绍 MobX 的主要组成部分以及每一部分涉及的重要 API。

10.2.1 state

state 是驱动应用的数据，是应用的核心。同 Redux 类似，我们依然可以把 state 分为三类：与领域直接相关的领域状态数据、反映应用行为（登录状态、当前是否有 API 请求等）的应用状态数据和代表 UI 状态的 UI 状态数据。在实际使用中，一般还会另外创建一个 store 来管理 state，这和 Redux 中的 store 也是类似的。但 MobX 中，可以在一个应用中使用多个 store，store 中的 state 也是可变的。另外，MobX 的 state 的结构不需要做标准化处理（Normalize），可以有多层嵌套结构，以方便 UI 组件使用为指导原则，这也是和 Redux 的 state 不同的地方。

MobX 提供了 observable 和@observable 两个 API 创建可观测的 state，用法如下：

```
observable(value)

@observable classProperty = value
```

这两个 API 几乎可以用在所有的 JS 数据类型上。但根据不同类型的值创建出的可观测 state 的表现行为是有不同点的：

1. 普通对象（Plain Object）

普通对象指原型不存在或原型是 Object.prototype 的对象，例如，var obj={"book":"react"}、var obj = new Object({"book":"react"})都是普通对象。MobX 根据普通对象创建一个可观测的新对象，新对象的属性和普通对象相同，但每一个属性都是可观测的，例如：

```
import { observable, autorun } from "mobx";

var person = observable({
```

```
  name: "Jack",
  age: 20
});

// mobx.autorun 会创建一个 reaction 自动响应 state 的变化，后面将会介绍
autorun(() =>
  console.log(`name:${person.name}, age:${person.age}`)
);

person.name = "Tom";
// 输出：name:Tom, age:20

person.age = 25;
// 输出：name:Tom, age:25
```

person 的 name 和 age 属性都是可观测的，任意属性的变化都会触发 autorun 的重新执行。使用@observable 可以将代码改写如下：

```
import { observable, autorun } from "mobx";

class Person {
  @observable name = "Jack";
  @observable age = 20;
}

var person = new Person();

autorun(() =>
  console.log(`name:${person.name}, age:${person.age}`)
);

person.name = "Tom";
// 输出：name:Tom, age:20

person.age = 25;
// 输出：name:Tom, age:25
```

使用普通对象转换成可观测对象时，还需要注意下面几个问题：

- 只有当前普通对象已经存在的属性才会转换成可观测的，后面添加的新属性都不会自动变成可观测的，例如：

  ```
  import { observable, autorun } from "mobx";

  var person = observable({
  ```

```
  name: "Jack",
  age: 20
});

autorun(() =>
  console.log(`name:${person.name}, age:${person.age},
address:${person.address}`)
);

person.address = "Shanghai";
// 没有新的输出
```

address 是后来添加的属性，它的改变并不会引起 autorun 的重新执行。

● 属性的 getter 会自动转换成 computed value，效果和使用@computed 相同。

```
import { observable, autorun } from "mobx";

var person = observable({
  name: "Jack",
  age: 20,

  //自动转换成 computed value
  get labelText() {
    return `name:${this.name}, age:${this.age}`;
  }
});

autorun(() => console.log(person.labelText));

person.name = "Tom";
// 输出: name:Tom, age:20

person.age = 25;
// 输出: name:Tom, age:25
```

labelText 是一个 getter 方法，会自动转换成 computed value，autorun 中使用到了 labelText，labelText 的计算值又依赖于 name 和 age，所以 name 和 age 的改变会导致 autorun 重新执行。

● observable 会递归地遍历整个对象，每当遇到对象的属性值还是一个对象时（不包含非普通对象），这个属性值将会继续被 observable 转换，例如:

```
import { observable, autorun } from "mobx";

var person = observable({
  name: "Jack",
```

```
    address: {
      province: "Shanghai",
      district: "Pudong"
    }
  });

  autorun(() =>
    console.log(`name:${person.name},
  address:${JSON.stringify(person.address)}`)
  );

  person.address.district = "Xuhui";
  // 输出：name:Jack, address:{"province":"Shanghai","district":"Xuhui"}
```

person 的 name 和 address 属性是可观测的，address 的值是一个对象，因此会继续被 observable 处理，address 的 province 和 district 属性也被转换成可观测的。所以，当 person.address.district = "Xuhui"执行后，district 的改变也会导致 autorun 的重新执行。

此外，如果以后再给可观测属性赋新值并且新值是一个对象（不包含非普通对象）时，新值也会自动被转换成可观测的，例如：

```
import { observable, autorun } from "mobx";

var person = observable({
  name: "Jack",
  address: {
    province: "Shanghai",
    district: "Pudong"
  }
});

autorun(() =>
  console.log('name:${person.name}, address:${JSON.stringify
(person.address)}')
);

// 给可观测属性 address 赋新值
person.address = {
  province: "Beijing",
  district: "Xicheng"
};
// 输出：name:Jack, address:{"province":"Beijing","district":"Xicheng"}

person.address.district = "Dongcheng";
// 输出：name:Jack, address:{"province":"Beijing","district":"Dongcheng"}
```

person 的 address 被赋予一个新对象时，新对象被自动转换成可观测对象，因此，新对象的 district 属性发生改变后，autorun 依然会被触发。

2. ES 6 Map

返回一个新的可观测的 Map 对象。Map 对象的每一个对象都是可观测的，而且向 Map 对象中添加或删除新元素的行为也是可观测的，这也是 Map 类型的可观测 state 最大的特点，例如：

```
import { observable, autorun } from "mobx";

// Map 可以接收一个数组作为参数，数组的每一个元素代表 Map 对象中的一个键值对
var map = new Map([["name", "Jack"], ["age", 20]]);
var person = observable(map);

autorun(() =>
  console.log(`name:${person.get("name")}, age:${person.get("age"),
address:${person.get("address")}`)
);

person.set("address", "Shanghai");
// 输出: name:Jack, age:20, address:Shanghai
```

person 是一个可观测的 Map 对象，当通过 Map 的 API 向 person 中添加新元素 address 时，autorun 会重新执行。

3. 数组

返回一个新的可观测数组。数组元素的增加或减少都会自动被观测，例如：

```
import { observable, autorun } from "mobx";

var todos = observable(["Learn React", "Learn Redux"]);

autorun(() => console.log(`待办事项数量: ${todos.length}`));

todos.push("Learn MobX");
// 输出: 待办事项数量: 3

todos.shift();
// 输出: 待办事项数量: 2
```

observable 作用于数组类型时，也会递归地作用于数组中的每个元素对象，处理规则和处理普通对象时的规则相同，例如：

```
import { observable, autorun } from "mobx";

var todos = observable([
```

```
  { text: "Learn React", finished: false },
  { text: "Learn Redux", finished: false }
]);

autorun(() => console.log(`todo 1 : ${todos[0].text}, finished:
${todos[0].finished}`));

todos[0].finished = true;
// 输出：todo 1 : Learn React, finished: true
```

todos 数组中的元素也转换成可观测对象，因此，元素属性的变化会导致 autorun 的重新执行。

4. 非普通对象

这里，非普通对象的概念是针对普通对象而言的，特指以自定义函数作为构造函数创建的对象。observable 会返回一个特殊的 boxed values 类型的可观测对象。注意，返回的 boxed values 对象并不会把非普通对象的属性转换成可观测的，而是保存一个指向原对象的引用，这个引用是可观测的。对原对象的访问和修改需要通过新对象的 get()和 set()方法操作，例如：

```
import { observable, autorun } from "mobx";

function Person(name, age) {
  this.name  = name;
  this.age = age;
}
var person = observable(new Person("Jack", 20));

// person 是 boxed values 类型，必须通过 get()才能获取到原对象
autorun(() => console.log(`name:${person.get().name},
age:${person.get().age}`));

person.get().age = 25
// 没有输出，因为 person 对象的属性不可观测

// person 封装的对象设置为一个新对象，引用发生变化，可观测
person.set(new Person("Jack", 20));
// 输出：name:Jack, age:20
```

将非普通对象的属性转换成可观测的是自定义构造函数的责任。正确的实现方式是：

```
import { extendObservable, autorun } from "mobx";

function Person(name, age) {
  // 使用 extendObservable 在构造函数内创建可观测属性
  extendObservable(this, {
    name: name,
```

```
    age: age
  });
}
var person = new Person("Jack", 20);

autorun(() => console.log(`name:${person.name}, age:${person.age}`));

person.age = 25;
// 输出: name:Jack, age:25
```

改写成使用装饰器@observable 的方式：

```
import { observable, autorun } from "mobx";

class Person {
  @observable name;
  @observable age;

  constructor(name, age) {
    this.name = name;
    this.age = age
  }
}

var person = new Person("Jack", 20);

autorun(() => console.log(`name:${person.name}, age:${person.age}`));

person.age = 25;
// 输出: name:Jack, age:25
```

5. 基本数据类型

MobX 是将包含值的属性（引用）转换成可观测的，而不是直接把值转换成可观测的。当 observable 接收的参数是 JavaScript 的基本数据类型时，MobX 不会把它们转换成可观测的，而是同处理非普通对象一样，返回一个 boxed values 类型的对象，例如：

```
import { observable, autorun } from "mobx";

// Jack 这个值是不可观测的，可观测的是指向这个对象的引用
const name = observable("Jack");

autorun(() => console.log(`name:${name.get()}`));

name.set("Tom");
```

```
// 输出：name:Tom
```

除了直接使用 observable 创建可观测对象外，还可以使用语义更加精确的 API 创建不同类型的可观测对象，例如：

```
observable.object(value)   //创建一个可观测的 Object
observable.array(value)    //创建一个可观测的 Array
observable.map(value)      //创建一个可观测的 Map
observable.box(value)      //创建一个可观测的 Boxed value
```

observable(value)相当于以上 API 的简写方式，会自动根据参数类型的不同使用不同的转换逻辑。

10.2.2　computed value

computed value 是根据 state 衍生出的新值，新值必须是通过纯函数计算得到的。computed value 依赖的 state 改变时，会自动重新计算，前提是这个 computed value 有被 reaction 使用。也就是说，computed value 采用延迟更新策略，只有被使用时才会自动更新。一般通过 computed 和@computed 创建 computed value，使用方式如下：

```
computed(() => expression)

@computed get classProperty() { return expression; }
```

computed 一般用于接收一个函数，例如：

```
import { observable, computed, autorun } from "mobx";

var person = observable.object({
  name: "Jack",
  age: 20,
});

// 使用 computed 函数创建 computed value
const isYoung = computed(()=> {
  return person.age < 25;
})

autorun(() =>
  console.log(`name:${person.name}, isYoung:${isYoung}`)
);

person.age = 25;
// 输出：name:Jack, isYoung:false
```

@computed 一般用于修饰 class 的属性的 getter 方法，例如：

```
import { observable, computed, autorun } from "mobx";

class Person {
  @observable name;
  @observable age;

  // 使用@computed 装饰器创建 computed value
  @computed get isYoung() {
    return this.age < 25;
  }

  constructor(name, age) {
    this.name = name;
    this.age = age
  }
}

var person = new Person("Jack", 20);

autorun(() => console.log(`name:${person.name},
isYoung:${person.isYoung}`));

person.age = 25;
// 输出: name:Jack, isYoung:false
```

10.2.3 reaction

reaction 是自动响应 state 变化的有副作用的函数。和 computed value 相同的地方是，它们都会因为 state 的变化而自动触发，所以 computed value 和 reaction 在 MobX 中都被称为 derivation（衍生）。derivation 是指可以从 state 中衍生出来的任何东西，例如值或者动作。与 computed value 不同的是，reaction 产生的不是一个值，而是执行一些有副作用的动作，例如打印信息到控制台、发送网络请求、根据 React 组件树更新 DOM 等。

使用 observer/@observer 封装 React 组件是常用的创建 reaction 的方式。observer/@observer 是 mobx-react 这个包提供的 API，常用的使用方式有如下三种：

```
observer((props, context) => ReactElement)
observer(class MyComponent extends React.Component { ... })
@observer class MyComponent extends React.Component { ... }
```

observer 的参数可以是一个 React 函数组件，也可以是一个 React 类组件，但对于类组件一般习惯使用@observer 创建 reaction。observer/@observer 本质上是将组件的 render 方法转换成 reaction，当 render 依赖的 state 发送变化时，render 方法会被重新调用。

除了 observer/@observer 外，常用的创建 reaction 的 API 还有 autorun、reaction、when，这几个 API 直接作用于函数而不是组件。

1. autorun

用法：

```
autorun(() => { sideEffect })
```

autorun 在前面的例子中已经多次使用到。使用 autorun 时，它接收的函数会被立即触发执行一次，以后的执行就依赖于函数使用的 state 的变化了。autorun 会返回一个清除函数 disposer，当不再需要观察相关 state 的变化时，可以调用 disposer 函数清除副作用，例如：

```
var numbers = observable([1,2,3]);
var sum = computed(() => numbers.reduce((a, b) => a + b, 0));

var disposer = autorun(() => console.log(sum.get()));
// 输出：6
numbers.push(4);
// 输出：10

disposer();  // 清除 autorun
numbers.push(5);
// 没有输出
```

2. reaction

用法：

```
reaction(() => data, data => { sideEffect }, options?)
```

它接收两个函数，第一个函数返回被观测的 state，这个返回值同时是第二个函数的输入值，只有第一个函数的返回值发生变化时，第二个函数才会被执行。第三个参数 options 是可选参数，提供一些可选设置，一般很少用到。reaction 也会返回一个清除函数 disposer。可见，相较于 autorun，reaction 可以对跟踪哪些对象有更多的控制。下面是一个示例：

```
const todos = observable([
  {
    title: "Learn React",
    done: true
  },
  {
    title: "Learn MobX",
    done: false
  }
]);

// 错误用法：只响应 todos 数组长度的变化，不会响应 title 属性的变化
const reaction1 = reaction(
  () => todos.length,
```

```
  length => console.log("reaction 1:", todos.map(todo => todo.title).join(", "))
);

// 正确用法：同时响应 todos 数组长度和 title 属性的变化
const reaction2 = reaction(
  () => todos.map(todo => todo.title),
  titles => console.log("reaction 2:", titles.join(", "))
);

todos.push({ title: "Learn Redux", done: false });
// 输出：
// reaction 2: Learn React, Learn MobX, Learn Redux
// reaction 1: Learn React, Learn MobX, Learn Redux

todos[0].title = "Learn Something";
// 输出：
// reaction 2: Learn Something, Learn MobX, Learn Redux
```

3. when

用法：

```
when(() => condition, () => { sideEffect })
```

condition 会自动响应它使用的任何 state 的变化，当 condition 返回 true 时，函数 sideEffect 会执行，且只执行一次。when 也会返回一个清除函数 disposer。when 非常适合用在以响应式的方式执行取消或清除逻辑的场景，例如：

```
class MyResource {
  constructor() {
    when(
      () => !this.isVisible,
      () => this.dispose()
    );
  }

  @computed get isVisible() {
    // 判断某个元素是否可见
  }

  dispose() {
    // 清除逻辑
  }
}
```

10.2.4　action

action 是用来修改 state 的函数。MobX 提供了 API action 和@action 用来包装 action 函数，但这并不是必需的。当 MobX 运行在严格模式下（调用 mobx.useStrict(true)即可启动严格模式）时，必须使用这两个 API 包装 action 函数。常见的用法有：

```
action(fn)
@action classMethod
```

为了让代码更加清晰可读，建议创建 action 函数时都要使用 action/@action。此外，action/@action 还能带来性能的提升，当函数内多次修改 state 时，action/@action 会执行批处理操作，只有所有的修改都执行完成后，才会通知相关的 computed value 和 reaction。下面是一个获取 BBS 帖子列表的 action：

```
@action fetchPostList(url) {
  this.pendingRequestCount++;
  return fetch(url).then(
    action(data => {
      this.pendingRequestCount--;
      this.posts.push(data);
    })
  );
}
```

这里需要注意，我们使用了两次 action 函数，因为 fetch 是异步执行的，执行完成的回调函数中也会修改 state，所以需要单独使用一个 action 包装回调函数。

使用 action 时，需要注意函数内 this 指向的问题，例如：

```
class Ticker {
  @observable tick = 0

  @action
  increment() {
    this.tick++;
  }
}

const ticker = new Ticker()
setInterval(ticker.increment, 1000) // 报错
```

在上面的例子中，increment 执行时，this 指向的是全局的 window 对象，并不是期望的 Ticker 的实例对象。可以使用箭头函数解决 this 绑定的问题：

```
class Ticker {
  @observable tick = 0;
```

```
  @action
  increment = () => {
    this.tick++;
  }
}

const ticker = new Ticker();
setInterval(ticker.increment, 1000);
```

此外，MobX 还提供了@action.bound 和 action.bound 两个 API 帮助完成 this 绑定的工作：

```
class Ticker {
  @observable tick = 0

  @action.bound
  increment() {
    this.tick++;
  }
}

const ticker = new Ticker();
setInterval(ticker.increment, 1000);
```

或

```
const ticker = observable({
  tick: 1,
  increment: action.bound(function() {
    this.tick++;
  })
})

setInterval(ticker.increment, 1000);
```

10.3 MobX 响应的常见误区

一般情况下，MobX 会按照我们的预期进行工作，但在一些场景下，如果不真正理解 MobX 到底是对什么进行响应，就会写出错误的代码。这里总结了 3 个常见的误区：

（1）MobX 是通过追踪属性的访问来追踪值的变化，而不是直接追踪值本身的变化。所以，必须在 MobX 的 derivation（computed value 和 reaction）中解引用（dereference）可观测对象的属性，才能正确观测到这些属性值的变化。例如：

```
var todo = observable({
```

```
  title: "Learn React"
})

autorun(() => {
    console.log(todo.title)
})

todo = observable({ title: "Bar" })
```


注 意

解引用（dereference）指根据引用获取引用指向的值的过程。它和引用（reference）过程是两个相反的过程。引用过程可以记作：reference of B => A，解引用过程可以记作：dereference of A => B。

autorun 不会有响应。todo 虽然被改变了，但它只是一个指向一个可观测对象的变量（引用），它本身并不是可观测的。正确的写法是：

```
var todo = observable({
  title: "Learn React"
})

autorun(() => {
    console.log(todo.title)
})

todo.title = "Bar"
```

在 autorun 这个 reaction 中解引用 title 属性的值（通过 todo.title 的访问方式解引用），所以 title 的变化可以被正确追踪。

再考虑下面的例子：

```
var todo = observable({
  title: "Learn React"
})
var title = todo.title;

autorun(() => {
  console.log(title)
})

title = "Bar"
```

autorun 不会有响应。todo.title 在 autorun 外部被解引用，autorun 内部使用的 title 变量只是一个字符串类型的值，是不可观测的。MobX 必须通过追踪 todo.title 来追踪 title 属性值的变化。

最后一个例子：

```
var todo = observable({
  task: {
    title: "Learn React",
    content: "Read more books about React"
  }
})
var task = todo.task;

autorun(() => {
  console.log(task.title)
})

todo.task.title = "Bar";  // 修改 1
todo.task = {             // 修改 2
  title: "Learn MobX",
  content: "Read more books about MobX"
};
```

这个例子更具有迷惑性。修改 1 会触发 autorun 响应，修改 2 不会触发 autorun 响应。原因是 todo.task 和 task 变量是指向同一个可观测对象的引用，在 autorun 内部解引用 task.title，修改 1 对 title 属性的修改当然可以被 autorun 追踪到；修改 2 改变的是 todo 的 task 属性，在 autorun 内部并没有解引用 task 属性，所以 task 属性值的变化无法被 autorun 追踪到。

（2）MobX 只追踪同步执行过程中的数据。

例如：

```
var todo = observable({
  title: "Learn React"
});

autorun(() => {
  setTimeout(() => console.log(todo.title), 100);
});

todo.title = "Bar";
```

autorun 不会有响应。autorun 执行期间并没有访问任何可观测对象，todo 是在 setTimeout 异步执行期间访问的。

（3）observer 创建的组件，只有当前组件 render 方法中直接使用的数据才会被追踪，例如：

```
const MyComponent = observer(({ todo }) =>
  <SomeContainer
    title = {() => <div>{todo.title}</div>}
  />
```

```
)

// 注意，这只是个示例，修改 todo.title 的正确方式是在 MyComponent 的父组件中完成
todo.title = "Bar"  // 组件不会重新渲染
```

这个例子中，SomeContainer 组件的 title 是一个回调函数，用于渲染 title。虽然看似 todo.title 是在 MyComponent 的 render 方法中使用的，但并不是直接使用的，因为回调函数 title 的执行是在 SomeContainer 内，回调函数 title 执行时，todo.title 才是直接使用的。要想让 SomeContainer 可以正确响应 todo.title 的变化，SomeContainer 本身也需要使用 observer 包装。

如果 SomeContainer 来自外部库，就不方便直接使用 observer 包装 SomeContainer。这时候，SomeContainer 的 title 回调函数中可以使用一个可观测的组件，响应 todo.title 的变化：

```
const MyComponent = observer(({ todo }) =>
  <SomeContainer
    title = {() => <TitleRenderer todo={todo} />}
  />
)

// TitleRenderer 是一个可观测组件，SomeContainer 通过使用 TitleRenderer，
// 响应 title 的变化
const TitleRenderer = observer(({ todo }) =>
  <div>{todo.title}</div>)
)

todo.title = "Bar"  // 组件会重新渲染
```

还有另一种方案是使用 mobx-react 包提供的 Observer 组件，它不接收参数，只需要单个 render 函数作为子节点：

```
import { Observer } from "mobx-react";

const MyComponent = ({ todo }) =>
  <SomeContainer
    title = {() =>
      <Observer>
        {() => <div>{todo.title}</div>}
      </Observer>
    }
  />

todo.title = "Bar"  // 组件会重新渲染
```

10.4 在 React 中使用 MobX

MobX 提供了一个 mobx-react 包帮助开发者方便地在 React 中使用 MobX。我们在前面已经多次使用的 observer/@observer 就来自这个包。下面介绍 mobx-react 中另外两个常用 API。

- Provider
  Provider 是一个 React 组件，利用 React 的 context 机制把应用所需的 state 传递给子组件。它的作用与 react-redux 提供的 Provider 组件是相同的。
- inject
  inject 是一个高阶组件，它和 Provider 结合使用，用于从 Provider 提供的 state 中选取所需数据，作为 props 传递给目标组件。常用方式有如下两种:

```
inject("store1", "store2")(observer(MyComponent))

@inject("store1", "store2") @observer MyComponent
```

一个简单示例:

```
import React, { Component } from "react";
import ReactDOM from "react-dom";
import { observer, inject, Provider } from "mobx-react";
import { observable } from "mobx";

@observer
@inject("store")  // inject 从 context 中取出 store 对象，注入到组件的 props 中
class App extends Component {
  render() {
    const { store } = this.props;
    return (
      <div>
        <ul>
          {store.map(todo => <TodoView todo={todo} key={todo.id} />)}
        </ul>
      </div>
    );
  }
}

const TodoView = observer(({ todo }) => {
  return <li>{todo.title}</li>;
});
```

```
// 构造 store 及其初始数据
const todos = observable([]);
todos.push({ id: 1, title: "Task1" });
todos.push({ id: 2, title: "Task2" });

ReactDOM.render(
  {/* Provider 向 context 中注入 store 对象 */}
  <Provider store={todos}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

10.5　本章小结

本章介绍了 MobX 的思想、主要组成和基本用法。MobX 同样遵循单项数据流，通过 action 改变 state，state 的变化又会触发 computed value 和 reaction 的重新执行。可观测对象是使用 MobX 的核心，读者要理解如何创建可观测对象、不同类型的可观测对象的表现行为有何差异以及使用可观测对象的几个常见误区。MobX 提供了 mobx-react 包，用于方便地在 React 应用中使用 MobX。

下一章，我们将在实战项目中使用 MobX。

第 11 章

MobX 项目实战

本章将使用 MobX 作为状态管理方案重构 BBS 项目。本章项目源代码的目录为/chapter-11/bbs-mobx。

11.1 组织项目结构

使用 MobX 时，没有必要像使用 Redux 那样区分容器组件和展示组件。所有组件会自动根据 state 的变化进行渲染（当然，前提是组件使用 observer/@observer 包装），所有组件都相当于展示组件。

这里读者可能会有疑问：MobX 项目中，所有组件都需要使用 observer/@observer 包装吗？这倒也不是，如果组件中使用到可观测的 state，组件就必须使用 observer/@observer 包装；否则可以不使用 observer/@observer。但即使组件中没有使用可观测的 state，使用 observer/@observer 包装组件也是有好处的，因为 observer/@observer 会将组件使用的不可观测的 props 转换成可观测的 props，这样只有当 props 真正发生改变时，当前组件才会重新渲染。简单理解的话，observer/@observer 的使用相当于重写了组件的 shouldComponentUpdate 方法。所以可以在项目中尽可能多地使用 observer/@observer，这可以提高所有组件的渲染效率。当然，这种方式也有一个缺点：难以在不使用 MobX 的项目中复用这些组件。

我们前面已经提到，MobX 中的 state 一般会封装在不同的 store 中，store 不仅保存了 state，还保存了操作 state 的方法。对于与领域直接相关的 state，一般会创建专门的 model 实体类，用于描述 state。

根据上面的分析，BBS 的项目结构如图 11-1 所示。

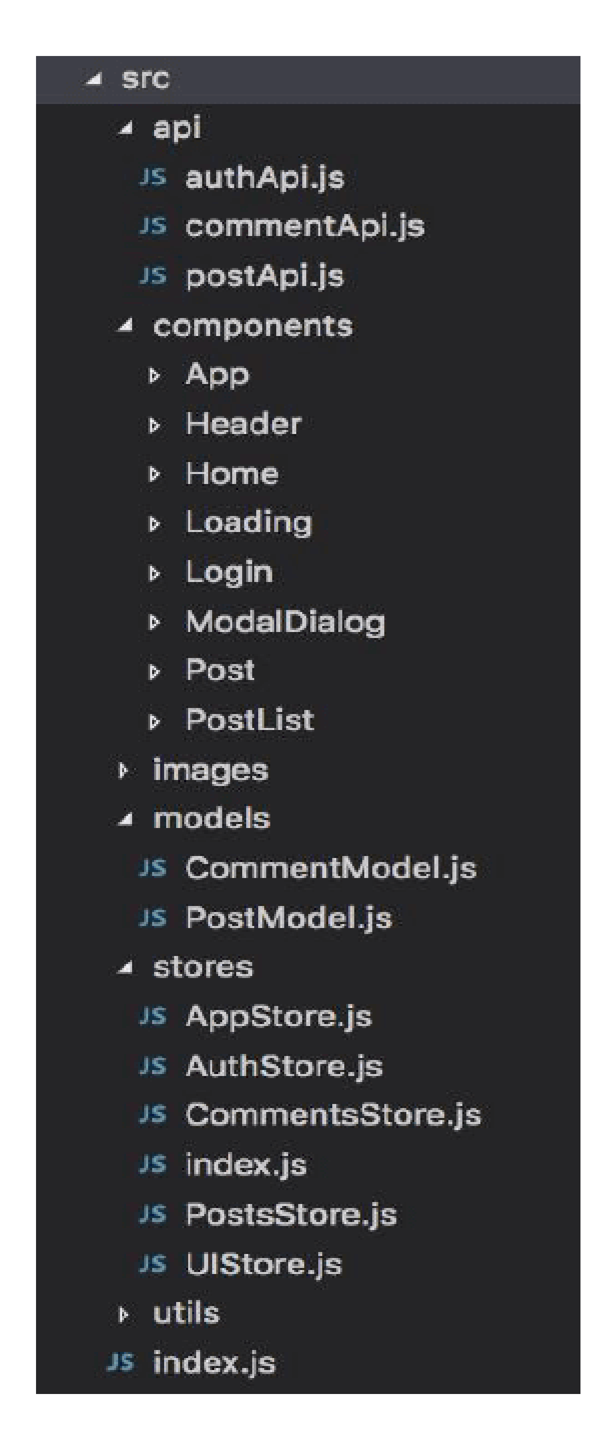


图 11-1

这里，我们还将 store 中使用到的网络请求单独封装，放到 api 文件夹下，这样有助于测试 store 时模拟网络请求。

11.2　设计 store

Store 的职责是将组件使用的业务逻辑和状态封装到单独的模块中，这样组件就可以专注于 UI 渲染。第 10 章介绍 state 时已经提到，state 可以分为三类：与领域直接相关的领域状态数据、反映应用行为（登录状态、当前是否有 API 请求等）的应用状态数据和代表 UI 状态的 UI 状态数据。后两种 state 一般不会涉及太多逻辑，仅仅是关于应用、UI 的一些松散状态的读取和简单修改，封装这两种 state 的 store 实现也很直观。领域 state 的数据结构较复杂，且往往涉及较多的逻辑处理。领域 state 可以使用普通对象来描述，也可以使用 class 来描述，例如描述一个待办任务 todo：

```
// 使用普通对象
var todo = {id: 1, title: "Todo1", finished: false}

// 使用 class
class Todo {
  id;
  title;
  finished;
}
```

使用 class 比使用普通对象描述 state 有一些优势：

（1）class 内可以定义方法，可以自己保存上下文信息而不依赖外部，因此 class 描述的 state 比普通对象描述的 state 更容易被单独使用。

（2）class 内可以方便地混合使用可观测属性和非可观测属性，例如，在 Todo class 中，我们希望只有 title 和 finished 是可观测的，那么只需要在这两个属性前使用@observable，id 继续作为不可观测属性使用。

（3）class 描述的 state 辨识度高且容易进行类型校验。

所以，稍复杂的领域 state 都建议大家使用 class 来描述。

一个领域 store 对应应用中的一个简单的领域概念，这个领域的 state 和 state 的管理都由这个 store 负责。具体来讲，领域 store 的职责有：

（1）实例化领域 state，并且保证领域 state 知道它属于哪一个 store。

（2）每一个领域 store 在应用中只能有一个实例对象，例如，应用中不能有两个 todoList store。

（3）更新领域 state，无论是通过服务器端获取，还是来自纯客户端的修改。

根据上面的介绍，我们可以为 BBS 创建 5 个 store：AppStore、AuthStore、UIStore、PostsStore、CommentsStore。AppStore 和 AuthStore 是应用状态 store，UIStore 是 UI store，PostsStore 和 CommentsStore 是领域 store。另外，还可以创建 PostModel 和 CommentModel 两个 class，代表领域 state。下面就来逐一分析每个 store。

1. AppStore

AppStore 管理的 state 包括应用当前的请求数量 requestQuantity 和应用的错误信息 error：

```
class AppStore {
  @observable requestQuantity = 0;
  @observable error = null;

  // ...
}
```

requestQuantity 间接决定界面上是否需要显示 Loading 效果，因此可以创建一个 computed value 直接标识是否需要显示 Loading 效果：

```
class AppStore {
  @computed get isLoading() {
    return this.requestQuantity > 0;
  }

  // ...
}
```

最后，为 AppStore 添加修改 state 的 action，完整代码如下：

```
import { observable, computed, action } from "mobx";
```

```
class AppStore {
  @observable requestQuantity = 0;
  @observable error = null;

  @computed get isLoading() {
    return this.requestQuantity > 0;
  }

  // 当前进行的请求数量加 1
  @action increaseRequest() {
    this.requestQuantity ++;
  }

  // 当前进行的请求数量减 1
  @action decreaseRequest() {
    if(this.requestQuantity > 0)
      this.requestQuantity --;
  }

  // 设置错误信息
  @action setError(error) {
    this.error = error;
  }

  // 删除错误信息，因为会作为回调函数被单独调用，所以这里需要绑定 this
  @action.bound removeError() {
    this.error = null;
  }
}

export default AppStore;
```

这里，removeError 使用@action.bound 绑定 this，因为存在 removeError 不是通过 AppStore 实例调用的场景，而是直接作为组件的回调函数被使用。为了保证 removeError 中的 this 一直指向的是 AppStore 的实例对象，所以使用了@action.bound。

2. AuthStore

AuthStore 负责用户的登录认证，使用到的 state 包括 userId、username 和 password，除了包含直接修改这几个 state 的 action 外，还定义了登录和注销两个 action，这两个 action 涉及网络请求，所以又会改变 AppStore 的 state，我们通过构造函数把 AppStore 的实例和登录 API 传递给 AuthStore，代码如下：

```
import { observable, action } from "mobx";

class AuthStore {
  api;
  appStore;
  @observable userId = sessionStorage.getItem("userId");
  @observable username = sessionStorage.getItem("username");
  @observable password = "";
  // 通过构造函数传递 AppStore 的实例对象和登录相关的 API
  constructor(api, appStore) {
    this.api = api;
    this.appStore = appStore;
  }

  @action setUsername(username) {
    this.username = username;
  }

  @action setPassword(password) {
    this.password = password;
  }

  @action login() {
    this.appStore.increaseRequest();
    const params = { username: this.username, password: this.password };
    // 异步回调函数需要单独定义成一个 action
    return this.api.login(params).then(action(data => {
      this.appStore.decreaseRequest();
      if (!data.error) {
        this.userId = data.userId;
        sessionStorage.setItem("userId", this.userId);
        sessionStorage.setItem("username", this.username);
        return Promise.resolve();
      } else {
        this.appStore.setError(data.error);
        return Promise.reject();
      }
    }));
  }

  @action.bound logout() {
    this.userId = null;
    this.username = null;
```

```
        this.password = null;
        sessionStorage.removeItem("userId");
        sessionStorage.removeItem("username");
      }
    }

    export default AuthStore;
```

3. UIStore

UIStore 负责新建帖子和编辑帖子两个 UI 状态的控制，代码很简单，不多做解释：

```
import { observable, action } from "mobx";

class UIStore {
  @observable addDialogOpen = false;
  @observable editDialogOpen = false;

  // 设置新建帖子编辑框的状态
  @action setAddDialogStatus(status) {
    this.addDialogOpen = status
  }

  // 设置修改帖子编辑框的状态
  @action setEditDialogStatus(status) {
    this.editDialogOpen = status
  }
}

export default UIStore;
```

4. PostsStore

PostsStore 负责帖子对应的领域，我们单独创建一个 class PostModel，用来描述帖子对应的 state：

```
import { observable, action } from "mobx";

class PostModel {
  store;  // PostModel 实例对象所属的 store
  id;
  @observable title;
  @observable content;
  @observable vote;
  @observable author;
  @observable createdAt;
```

```
  @observable updatedAt;

  constructor(store, id, title, content, vote, author, createdAt, updatedAt) {
    this.store = store;
    this.id = id;
    this.title = title;
    this.content = content;
    this.vote = vote;
    this.author = author;
    this.createdAt = createdAt;
    this.updatedAt = updatedAt;
  }

  // 根据 JSON 对象更新帖子
  @action updateFromJS(json) {
    this.title = json.title;
    this.content = json.content;
    this.vote = json.vote;
    this.author = json.author;
    this.createdAt = json.createdAt;
    this.updatedAt = json.updatedAt;
  }

  // 静态方法，创建新的 PostModel 实例
  static fromJS(store, object) {
    return new PostModel(
      store,
      object.id,
      object.title,
      object.content,
      object.vote,
      object.author,
      object.createdAt,
      object.updatedAt
    );
  }

}

export default PostModel;
```

PostModel 包含帖子标题、内容、作者等可观测属性，action updateFromJS 用于根据服务器端返回的数据更新 PostModel 实例，静态方法 fromJS 用于根据服务器端返回的数据构造 PostModel

的实例，这两个方法在 PostsStore 中都要用到。

PostsStore 中保存一个可观测的数组 posts，posts 的元素是 PostModel 的实例，PostsStore 的 action 包括获取帖子列表、获取帖子详情、新建帖子和修改帖子，代码如下：

```
import { observable, action, toJS } from "mobx";
import PostModel from "../models/PostModel";

class PostsStore {
  api;
  appStore;
  authStore;
  @observable posts = [];  // 数组的元素是 PostModel 的实例

  constructor(api, appStore, authStore) {
    this.api = api;
    this.appStore = appStore;
    this.authStore = authStore;
  }

  // 根据帖子 id 获取当前 store 中的帖子
  getPost(id) {
    return this.posts.find(item => item.id === id);
  }

  // 从服务器获取帖子列表
  @action fetchPostList() {
    this.appStore.increaseRequest();
    return this.api.getPostList().then(
      action(data => {
        this.appStore.decreaseRequest();
        if (!data.error) {
          this.posts.clear();
          data.forEach(post => this.posts.push(PostModel.fromJS(this, post)));
          return Promise.resolve();
        } else {
          this.appStore.setError(data.error);
          return Promise.reject();
        }
      })
    );
  }

  // 从服务器获取帖子详情
```

```
@action fetchPostDetail(id) {
  this.appStore.increaseRequest();
  return this.api.getPostById(id).then(
    action(data => {
      this.appStore.decreaseRequest();
      if (!data.error && data.length === 1) {
        const post = this.getPost(id);
        // 如果 store 中当前 post 已存在，就更新 post
        // 否则，添加 post 到 store
        if (post) {
          post.updateFromJS(data[0]);
        } else {
          this.posts.push(PostModel.fromJS(this, data[0]));
        }
        return Promise.resolve();
      } else {
        this.appStore.setError(data.error);
        return Promise.reject();
      }
    })
  );
}

// 新建帖子
@action createPost(post) {
  const content = { ...post, author: this.authStore.userId, vote: 0 };
  this.appStore.increaseRequest();
  return this.api.createPost(content).then(
    action(data => {
      this.appStore.decreaseRequest();
      if (!data.error) {
        this.posts.unshift(PostModel.fromJS(this, data));
        return Promise.resolve();
      } else {
        this.appStore.setError(data.error);
        return Promise.reject();
      }
    })
  );
}

// 更新帖子
@action updatePost(id, post) {
```

```
    this.appStore.increaseRequest();
    return this.api.updatePost(id, post).then(
      action(data => {
        this.appStore.decreaseRequest();
        if (!data.error) {
          const oldPost = this.getPost(id);
          if (oldPost) {
            /* 更新帖子的 API，返回数据中的 author 只包含 authorId
             * 因此需要从原来的 post 对象中获取完整的 author 数据。
             * toJS 是 MobX 提供的函数，用于把可观测对象转换成普通的 JS 对象。 */
            data.author = toJS(oldPost.author);
            oldPost.updateFromJS(data);
          }
          return Promise.resolve();
        } else {
          this.appStore.setError(data.error);
          return Promise.reject();
        }
      })
    );
  }
}

export default PostsStore;
```

5. CommentsStore

CommentsStore 负责评论对应的领域，与 PostsStore 相同，先创建 class CommentModel，描述评论对应的 state：

```
import { observable } from "mobx";

class CommentModel {
  store;
  id;
  @observable content;
  @observable author;
  @observable createdAt;
  @observable updatedAt;

  constructor(store, id, content, author, createdAt, updatedAt) {
    this.store = store;
    this.id = id;
    this.content = content;
```

```
    this.author = author;
    this.createdAt = createdAt;
    this.updatedAt = updatedAt;
  }

  static fromJS(store, object) {
    return new CommentModel(
      store,
      object.id,
      object.content,
      object.author,
      object.createdAt,
      object.updatedAt
    );
  }
}

export default CommentModel;
```

CommentsStore 中保存一个可观测的数组 comments，comments 的元素是 CommentModel 的实例。CommentsStore 中定义的 action 包括获取某个帖子的评论列表和新建评论，代码如下：

```
import { observable, action } from "mobx";
import CommentModel from "../models/CommentModel";

class CommentsStore {
  api;
  appStore;
  authStore;
  @observable comments = [];   // 数组的元素是 CommentModel 的实例

  constructor(api, appStore, authStore) {
    this.api = api;
    this.appStore = appStore;
    this.authStore = authStore;
  }

  // 获取评论列表
  @action fetchCommentList(postId) {
    this.appStore.increaseRequest();
    return this.api.getCommentList(postId).then(action(data => {
      this.appStore.decreaseRequest();
      if (!data.error) {
        this.comments.clear();
        data.forEach(item => this.comments.push(CommentModel.fromJS(this,
item)));
```

```
        return Promise.resolve();
      } else {
        this.appStore.setError(data.error);
        return Promise.reject();
      }
    }));
  }

  // 新建评论
  @action createComment(content){
    this.appStore.increaseRequest();
    return this.api.createComment(content).then(action(data => {
      this.appStore.decreaseRequest();
      if (!data.error) {
        this.comments.unshift(CommentModel.fromJS(this, data));
        return Promise.resolve();
      } else {
        this.appStore.setError(data.error);
        return Promise.reject();
      }
    }));
  }
}

export default CommentsStore;
```

6. 合并 store

通常会把使用到的多个 store 再次合并成一个根 store，利用 mobx-react 提供的高阶组件 Provider 将根 store 注入组件树中。我们在 stores/index.js 中完成这项工作：

```
import AppStore from "./AppStore";
import AuthStore from "./AuthStore";
import PostsStore from "./PostsStore";
import CommentsStore from "./CommentsStore";
import UIStore from "./UIStore";
import authApi from "../api/authApi";
import postApi from "../api/postApi";
import commentApi from "../api/commentApi";

// 每个 store 在应用中只存在一个实例对象
const appStore = new AppStore();
const authStore = new AuthStore(authApi, appStore);
const postsStore = new PostsStore(postApi, appStore, authStore);
const commentsStore = new CommentsStore(commentApi, appStore, authStore);
const uiStore = new UIStore();
```

```
const stores = {
  appStore,
  authStore,
  postsStore,
  commentsStore,
  uiStore
};

export default stores;
```

在 stores/index.js 中，对每一个 store 都进行实例化，然后合并成一个对象导出。这样既保证每一个 store 只有一个实例，又便于外部组件的使用。

11.3 视图层重构

视图层的重构工作主要是使用 observer/@observer 将组件转换成一个个 reaction，同时使用 mobx-react 提供的 inject/@inject 注入组件所需的 store。

首先，在组件树的最外层使用 mobx-react 提供的 Provider 组件注入合并后的 store：

```
// index.js
import React from "react";
import ReactDOM from "react-dom";
import { useStrict } from 'mobx';
import { Provider } from "mobx-react";
import App from "./components/App";
import stores from "./stores";

// 在严格模式下，运行 MobX
useStrict(true);

ReactDOM.render(
  <Provider {...stores}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

组件 App 需要使用 appStore，主要代码如下：

```
@inject("appStore")  // @inject 注入使用的 Store: appStore
@observer  // @observer 把 App 组件转化成一个 reaction，自动响应 state 的变化
class App extends Component {
  // ...
```

```
  render() {
    const { error, isLoading, removeError } = this.props.appStore;
    const errorDialog = error && (
      <ModalDialog onClose={removeError}>{error.message ||
error}</ModalDialog>
    );

    return (
      <div>
        <Router>
          <Switch>
            <Route exact path="/" component={AsyncHome} />
            <Route path="/login" component={AsyncLogin} />
            <Route path="/posts" component={AsyncHome} />
          </Switch>
        </Router>
        {errorDialog}
        {isLoading && <Loading />}
      </div>
    );
  }
}

export default App;
```

再来看一下组件 PostList，它需要使用 postsStore、authStore 和 uiStore 三个 store，代码如下：

```
import React, { Component } from "react";
import { inject, observer } from "mobx-react";
import PostsView from "./PostsView";
import PostEditor from "../Post/PostEditor";
import "./style.css";

@inject("postsStore", "authStore", "uiStore")
@observer
class PostList extends Component {
  componentDidMount() {
    // 获取帖子列表
    this.props.postsStore.fetchPostList();
  }

  // 保存新建的帖子
  handleSave = data => {
    this.props.postsStore
      .createPost(data)
      .then(() => this.props.uiStore.setAddDialogStatus(false));
```

```
  };

  // 取消新建帖子的状态
  handleCancel = () => {
    this.props.uiStore.setAddDialogStatus(false);
  };

  // 设置新建帖子的状态
  handleNewPost = () => {
    this.props.uiStore.setAddDialogStatus(true);
  };

  render() {
    const { postsStore, authStore, uiStore } = this.props;
    return (
      <div className="postList">
        <div>
          <h2>帖子列表</h2>
          {authStore.userId ? (
            <button onClick={this.handleNewPost}>发帖</button>
          ) : null}
        </div>
        {uiStore.addDialogOpen ? (
          <PostEditor onSave={this.handleSave} onCancel={this.handleCancel} />
        ) : null}
        <PostsView posts={postsStore.posts} />
      </div>
    );
  }
}

export default PostList;
```

其他组件的改造也都类似，为节约篇幅，这里不再一一介绍，请读者参考源代码。

11.4　MobX 调试工具

mobx-react-devtools 是一个用于调试 MobX+React 项目的工具，它可以追踪组件的渲染以及组件依赖的可观测数据。我们为 BBS 项目添加这个调试工具：

```
// 安装
npm install mobx-react-devtools --save-dev
```

只需要在开发环境下使用调试工具，安装命令使用的参数是--save-dev。

将 mobx-react-devtools 提供的调试组件添加到 App 组件中：

```
@inject("appStore")
@observer
class App extends Component {
  renderDevTool() {
    // 在开发环境下，添加调试工具
    if (process.env.NODE_ENV !== "production") {
      const DevTools = require("mobx-react-devtools").default;
      return <DevTools />;
    }
  }

  render() {
    const { error, isLoading, removeError } = this.props.appStore;
    const errorDialog = error && (
      <ModalDialog onClose={removeError}>{error.message || error}
</ModalDialog>
    );

    return (
      <div>
        <Router>
          <Switch>
            <Route exact path="/" component={AsyncHome} />
            <Route path="/login" component={AsyncLogin} />
            <Route path="/posts" component={AsyncHome} />
          </Switch>
        </Router>
        {errorDialog}
        {isLoading && <Loading />}
        {/* 添加 MobX 调试组件 */}
        {this.renderDevTool()}
      </div>
    );
  }
}

export default App;
```

再次运行程序，界面右上角会多出三个小图标，这是 mobx-react-devtools 的功能键。点击第一个图标，当组件发生渲染行为时，对应的组件会被高亮显示，高亮组件右上角显示的 3 个数字分别代表截至当前组件渲染的次数、组件 render 方法执行的时间、组件从 render 方法开始到渲染到浏览器界面使用的时间，如图 11-2 所示。



图 11-2

点击第二个图标后，再用鼠标选择任意一个组件，可以查看该组件会对哪些数据的变化做出响应。图 11-3 显示的是一个 PostItem 组件实例所依赖的数据。

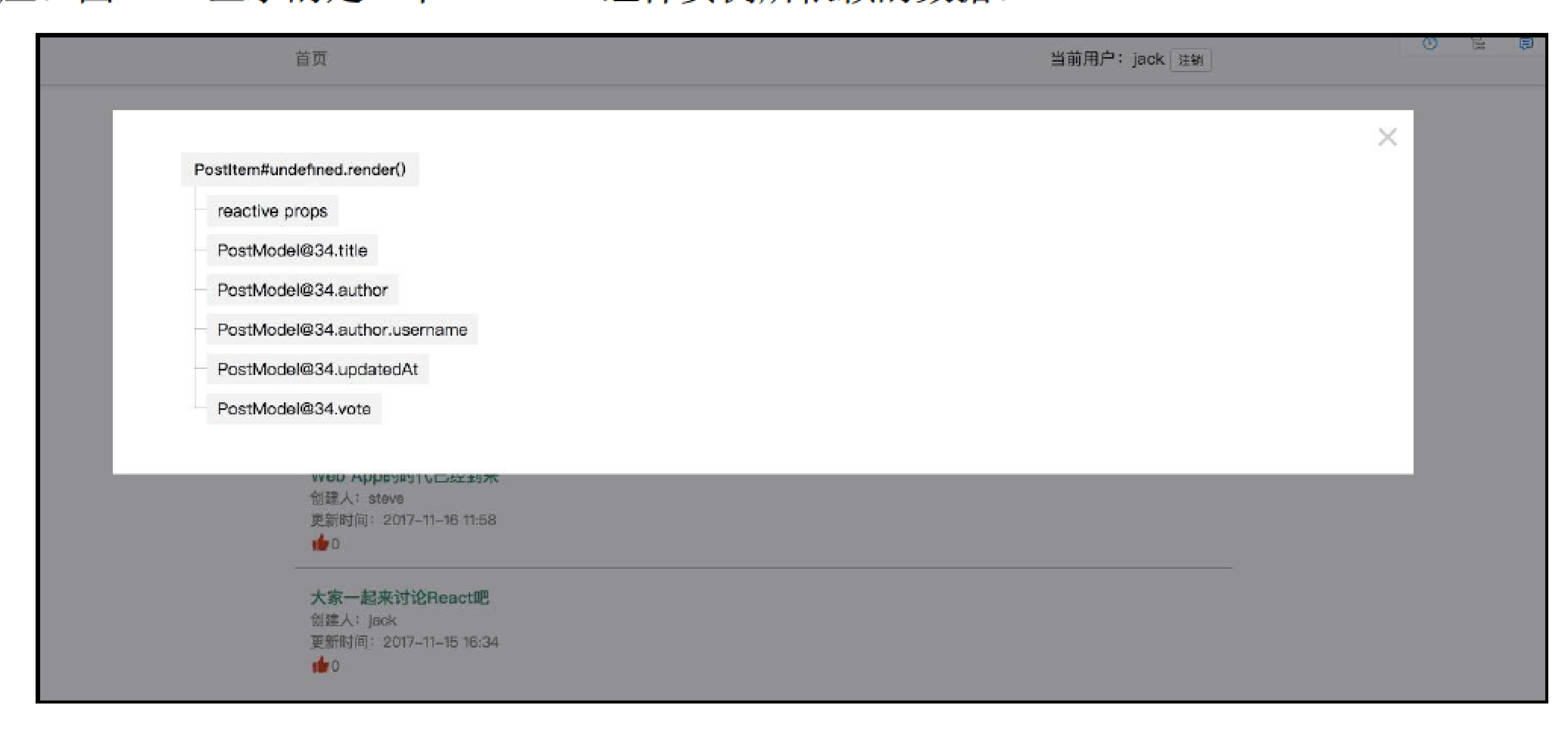


图 11-3

点击第三个图标，控制台会输出发生的 action、响应的 reaction 等调试日志信息。

11.5 优化建议

MobX 和 React 结合使用时，有一些常用的优化技巧可以帮助提高组件的渲染效率。

1. 尽可能多地使用小组件

observer/@observer 包装的组件会追踪 render 方法中使用的所有可观测对象，所以组件越小，组件追踪的对象越少，引起组件重新渲染的可能性也越小。

2. 在单独的组件中渲染列表数据

列表数据的渲染是比较耗费性能的，尤其是在列表数据量大的情况下，例如：

```
@observer
class MyComponent extends Component {
  render() {
    const { todos, user } = this.props;
    return (
      <div>
        {user.name}
        <ul>{todos.map(todo => <TodoView todo={todo} key={todo.id} />)}</ul>
      </div>
    );
  }
}
```

user.name 的改变会导致重新创建一个 TodoView 元素的数组，虽然这并不会导致重复渲染这些 TodoView，但 React 比较新旧 TodoView 元素的过程本身也很耗费性能。所以，更好的写法是：

```
@observer
class MyComponent extends Component {
  render() {
    const { todos, user } = this.props;
    return (
      <div>
        {user.name}
        <TodosView todos={todos} />
      </div>
    );
  }
}

@observer
class TodosView extends Component {
  render() {
    const { todos } = this.props;
    return <ul>{todos.map(todo => <TodoView todo={todo} key={todo.id} />)}
</ul>;
  }
}
```

3. 尽可能晚地解引用（dereference）对象属性

MobX 通过追踪对象属性的访问来追踪值的变化，所以在层级越低的组件中解引用对象属性，由这个属性的变化导致的重新渲染的组件的数量就越少。（只有解引用对象属性的组件及其子组件

会重新渲染）。例如：

```
// 方式 1
@observer
class DisplayName extends Component {
  render() {
    const {person} = this.props
    return <div>{person.name}<div/>
  }
}

class MyComponent extends Component {
  render() {
    const {person} = this.props
    return <DisplayName person={person} />
  }
}

// 方式 2
@observer
class DisplayName extends Component {
  render() {
    const {name} = this.props
    return <div>{name}<div/>
  }
}

class MyComponent extends Component {
  render() {
    const {person} = this.props
    return <DisplayName name={person.name} />
  }
}
```

person 是一个可观测对象，对于方式 1，当 person 的属性 name 发生变化时，DisplayName 会自动重新渲染，而不需要通过父组件 MyComponent 的重新渲染来触发。对于方式 2，DisplayName 使用的是 name 这个值，是不可观测的，因此，要想让 DisplayName 重新渲染，首先必须让 DisplayName 的父组件 MyComponent 重新渲染，这样就导致更多组件会发生重复渲染。

但方式 2 更容易理解，对于 DisplayName 组件仅需要接收 name 作为属性就足够了，接收 person 作为属性反而有些多余。为了兼顾效率和可读性，可以这样实现：

```
const PersonNameDisplayer = observer(({ props }) => <DisplayName
name={props.person.name} />)
```

这里新增了一个组件 PersonNameDisplayer，由这个组件负责渲染 DisplayName，PersonNameDisplayer 会自动响应 name 的变化，重新渲染 DisplayName 组件，同时 DisplayName

组件仍然只需要接收 name 属性即可。本质上还是使用小组件的思路进行优化。

4. 提前绑定函数

例如：

```
@observer
class MyComponent extends Component {
  render() {
    return <MyWidget onClick={() => { alert('hi') }} />
  }
}
```

这种写法，MyComponent 的 render 方法每次被调用时，MyWidget 的 onClick 属性的值都是一个新的函数，导致 MyWidget 的 render 方法一定会被重新调用，而无论其他属性是否发生变化，MobX 对组件渲染做的优化工作都会浪费。更好的写法是：

```
@observer
class MyComponent extends Component {
  handleClick = () => {
    alert('hi')
  }

  render() {
    return <MyWidget onClick={this.handleClick} />
  }
}
```

本节介绍的这几种优化方法虽然看似很基础，但读者真正掌握并理解这些优化方法后，相信会对 MobX 有更加深入的理解。

11.6　Redux 与 MobX 比较

在学习完 Redux 和 MobX 后，很多读者可能会不知道在项目中应该选择使用哪一个，毕竟这两个库都是非常优秀的状态管理解决方案。我们从以下几个方面对比 Redux 与 MobX。

1. Store

Redux 是单一数据源，整个应用共享一个 store 对象，而 MobX 可以使用多个 store。因此，MobX 可以将应用逻辑拆分到不同 store 中，而 Redux 需要通过拆分 reducer 来拆分应用逻辑。当应用越来越复杂时，单一 store 可以更方便地在不同组件间共享，而维护多 store 间的数据共享、相互引用关系会变得很复杂。

2. State

Redux 使用普通 JavaScript 对象存储 state，并且 state 是不可变的，每次状态的变更都必须重新创建一个新的 state。MobX 中的 state 是可观测对象，并且 state 是可以被直接修改的，state 的变化会自动触发使用它的组件重新渲染。此外，Redux 的 state 结构应该尽量扁平化，减少嵌套层级，而 MobX 的 state 结构可以任意嵌套，从这一点上来说，MobX 的 state 更容易直接被组件使用。

3. 编程范式

Redux 是基于函数式的编程思想，MobX 是面向对象的编程思想，因此，对于传统面向对象的开发者而言，MobX 更加友好。Redux 有严格的规范约束，而 MobX 更加灵活，开发者可以更加随意的编写代码。但对于大型项目来说，有严格的规范更易于后期的维护和扩展。

4. 代码量

因为 Redux 有严格的规范，所以往往需要写更多的代码来执行这些规范。例如，实现一个特性需要修改 action、reducer、组件等多个地方，而 MobX 只需要修改用到的 store 和视图组件。

5. 学习曲线

因为 MobX 的面向对象的编程思想以及没有太多规范约束，它的学习曲线更加平缓，更易于上手。对于不熟悉函数式编程的开发者，Redux 是比较难学习的，加上它的诸多规范限制，更增加了初学者的学习难度。

基于以上比较，一般建议当小型团队需要开发相对简单的应用时，可以选择使用 MobX，它易学习、上手快、代码量少；当团队规模较大或应用复杂度较高时，可以选择使用 Redux，它严格的规范有利于保障项目代码的可维护性和可扩展性。当然，技术的选择是没有绝对的，最终都需要根据实际业务场景做选择。

11.7 本章小结

本章结合项目实例，从项目结构的组织方式、store 的设计等方面详细介绍了如何在真实项目中使用 MobX，还介绍了 MobX 应用中常用的调试工具和常用的性能优化方法，最后对 Redux 与 MobX 进行了比较，希望通过比较能给读者在如何选择上提供一些建议。